PROJECTS ■ THEORY ■ APPLICATIONS ■ CIRCUITS ■ TECHNOLOGY

NUTS AND VOLTS

NUTS AND VOLTS
www.nutsvolts.com
September 2015
EVERYTHING FOR ELECTRONICS

Get Your PET OUT OF THE CEMETERY

It May Have A Few Colourful Lessons Yet To Teach!

Electrify Your Halloween Fun!

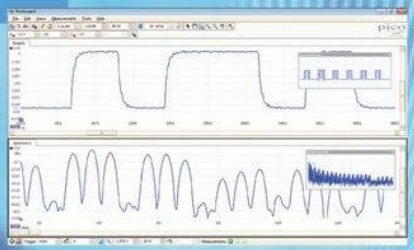
- Use a spectrum analyzer chip for voice syncing
- Build the remote control Little Book of Horror
- 13 Spooky Electronic Ideas to help your haunt
- Parallax Propeller Powered Pumpkin

- **Reducing Radio Frequency Interference**
  Step-by-Step guide to cleaning up your reception
- **Get Your AVR Microcontroller Talking**
  In-depth guide to speaking I2C on your ATmega

PC OSCILLOSCOPES

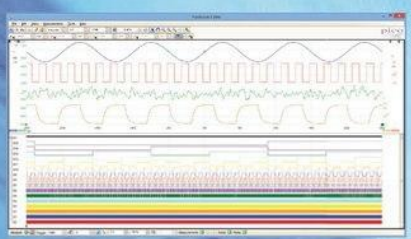
pico® Technology

Low cost



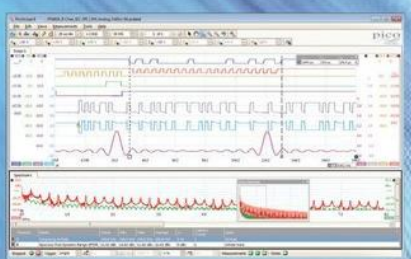
- 10 MHz to 200 MHz bandwidth
- 100 MS to 1 GS/s sampling
- 8 bit resolution (12 bit enhanced)
- 8 to 48 kS buffer memory
- USB powered
- Prices from $129

MSO



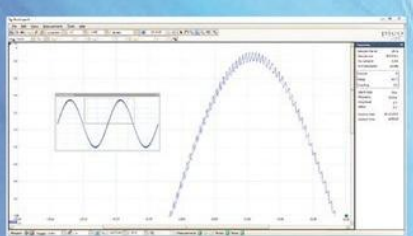
- 2 or 4 analog channels + 16 digital
- 50 to 200 MHz bandwidth
- 8 bit resolution (12 bit enhanced)
- 64 to 512 MS buffer memory
- USB or AC adaptor powered
- Prices from $819

Eight channels



- 20 MHz bandwidth
- 80 MS/s sampling
- 12 bit resolution (16 bit enhanced)
- 256 MS buffer memory
- USB powered
- Just $2305

Flexible resolution

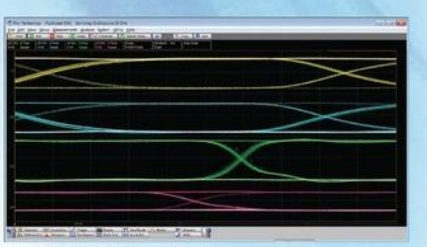


- 8, 12, 14, 15 & 16 bits all in one device
- 60 to 200 MHz bandwidth
- 250 MS/s to 1 GS/s sampling
- 8 to 512 MS buffer memory
- USB or AC adaptor powered
- Prices from $1155

2 GS memory

- 250 MHz to 1 GHz bandwidth
- 5 GS/s sampling
- 8 bit resolution (12 bit enhanced)
- 256 MS to 2 GS buffer memory
- AC adaptor powered
- Prices from $3295

20 GHz sampling



- DC to 20 GHz bandwidth
- 17.5 ps rise time
- 16 bit, 60 dB dynamic range
- AC adaptor powered
- Sig. gen, CDR, diff. TDR/TDT
- Prices from $14,995

Full software included as standard with serial bus decoding and analysis (CAN, LIN, RS232, I2C, I2S, SPI, FlexRay), segmented memory, mask testing, spectrum analysis, and software development kit (SDK) all as standard, with free software updates. Five years warranty real time oscilloscopes, 2 years warranty sampling oscilloscopes.

1-800-591-2796
www.picotech.com/pco543

The all new mikroBUS™ Cape

Turn your **BeagleBone Black** into anything you want with the **mikroBUS™ Cape**. With four sockets, you can pick and mix from over **130 click™ boards** to add all sorts of sensors, transceivers, encoders, connectors and many other modules to your projects. **New clicks are coming out weekly, so keep returning for more!**

MikroElektronika
DEVELOPMENT TOOLS | COMPILERS | BOOKS

www.mikroe.com/click/mikrobus-cape

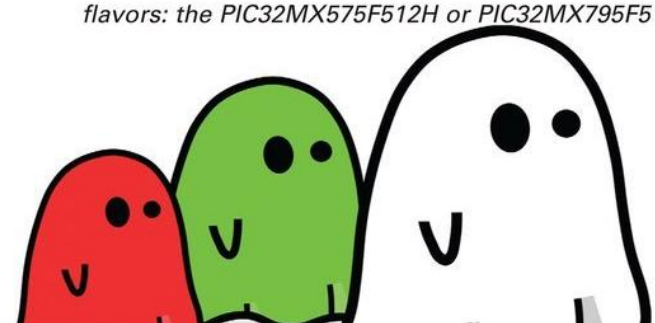
GET IT NOW

September 2015

SPECIAL HALLOWEEN PROJECTS

Departments

Columns



Nuts & Volts (ISSN 1528-9885/CDN Pub Agree #40702530) is published monthly for $26.95 per year by T & L Publications, Inc., 430 Princeland Court, Corona, CA 92879. PERIODICALS POSTAGE PAID AT CORONA, CA AND AT ADDITIONAL MAILING OFFICES. POSTMASTER: Send address changes to **Nuts & Volts, P.O. Box 15277, North Hollywood, CA 91615** or Station A, P.O. Box 54, Windsor ON N9A 6J5; cpcreturns@nutsvolts.com.

NUTS AND VOLTS

Published Monthly By
T & L Publications, Inc.
430 Princeland Ct.
Corona, CA 92879-1300
(951) 371-8497
FAX **(951) 371-3052**
Webstore orders only **1-800-783-4624**
www.nutsvolts.com

Subscription Orders
Toll Free **1-877-525-2539**
Outside US **1-818-487-4545**
P.O. Box 15277
North Hollywood, CA 91615

FOUNDER
Jack Lemieux

PUBLISHER
Larry Lemieux
publisher@nutsvolts.com

ASSOCIATE PUBLISHER/ ADVERTISING SALES
Robin Lemieux
robin@nutsvolts.com

EDITOR
Bryan Bergeron
techedit-nutsvolts@yahoo.com

VP OF OPERATIONS
Vern Graner
vern@nutsvolts.com

CONTRIBUTING EDITORS
Fred Eady
Lou Frenzel
Ward Silver
Steve Gray
Steven Bjork
Jon McPhalen
Tim Brown
Ron Hackett
Andrew Retallack
Bill Elwell
Steve Koci

CIRCULATION DEPARTMENT
subscribe@nutsvolts.com

SHOW COORDINATOR
Audrey Lemieux

WEB CONTENT
Michael Kaudze
website@nutsvolts.com

WEBSTORE MARKETING
Brian Kirkpatrick
sales@nutsvolts.com

WEBSTORE MANAGER
Sean Lemieux

ADMINISTRATIVE STAFF
Debbie Stauffacher
Re Gandara

DEVELOPING PERSPECTIVES

by Bryan Bergeron, Editor

Building Sandcastles in the Digital Sandbox

Developing hands-on expertise with microcontrollers has never been easier. I'm a fan of the Parallax (**www.Parallax.com**) series of educational kits and accompanying books, having used an early course featuring the BASIC Stamp. Parallax has since updated their offerings with Propeller Education kits, as well as an Arduino-based Board of Education shield. Parallax offers books for their proprietary Propeller kits, and there's no end of third-party support for the open source Arduino chip. The learning curve for the Propeller is steeper than that for the Arduino, but the payback is more processing power and flexibility.

Adafruit (**www.adafruit.com**) offers starter packs for the Arduino, BeagleBone Black, and Raspberry Pi. Like the Raspberry Pi, the BeagleBone Black isn't a typical microcontroller with a few hundred bytes of onboard storage. The BeagleBone Black is a single-board Linux computer with 4 GB of memory. At $55, it's about $20 more expensive than the Raspberry Pi and maybe double the price of an Arduino.

Even if you don't buy hardware from Adafruit, you should take a look at the example projects featured on their website. The photos and text are clear and easy to follow, and the projects are fun. They also start out very simply — as in making a BeagleBone Black or Raspberry Pi cause an LED to blink using Python. Once you've worked through LEDs, switches, servos, and the other basics, you can try your hand at driving graphic LCDs using the Python library, or use a Wi-Fi adapter to get a BeagleBone Black on the Internet.

One of the most interesting educational kits that I've run across lately is the SparkFun (**www.sparkfun.com**) Digital Sandbox. This Arduino-based kit is similar to some of the microcontroller boards offered by Parallax in that there are common components on the board to facilitate experimentation. The Digital Sandbox offers an LED bar graph, slide switch, temperature sensor, light sensor, RGB LED, potentiometer, microphone, and pushbutton. More than enough to start experimenting — all without having to solder anything. Not a big deal, but convenient. The kit sells for about $75.

Where the Digital Sandbox shines is in the use of the optional Ardublock graphical programming language. If you can draw an algorithm — as in block one feeds block two, and so on — you can program in Ardublock. The graphical Ardublock language reminds me of the free SCRATCH programming environment from MIT (**scratch.mit.edu**) in that it enables the younger set to access computing without having to code with IF-THEN-ELSE statements and the like.

Thanks to Ardublock, SparkFun recommends the Digital Sandbox for future microcontroller experts ages eight and up. Once you've mastered Ardublock, you can install the standard IDE (integrated development environment) and develop more complex programs.

So, there you have it. Microcontroller education for just about any age or budget. If funds are tight and you already own a few LEDs, switches, and sensors, I'd suggest an inexpensive Arduino clone and information from the pages of *Nuts & Volts*, Adafruit, or other web sources. If you're thinking of introducing microcontrollers to a class of eight or nine year olds, then try the Digital Sandbox — perhaps after an introduction to SCRATCH or other graphical programming environment. **NV**

In this column, Tim answers questions about all aspects of electronics, including computer hardware, software, circuits, electronic theory, troubleshooting, and anything else of interest to the hobbyist. Feel free to participate with your questions, comments, or suggestions. **Send all questions and comments to: Q&A@nutsvolts.com.**

- **Explanation of Radio Propagation**
- **Troubleshooting Tips**

Radio Propagation

Q I am in the sixth grade and am working on a project for school about how radios work. Can you help me by explaining how the radio signals can be received through the air?

Julie Gutierrez
Redding, CA

A Sixth grade is a great time to start learning about one of the most fascinating technologies of our times. Radio signals are traveling electromagnetic waves which can move through air or even the relatively empty reaches of outer space (**Figure 1**). Radio waves are used to carry information (such as music, voice, telephone, computer data) from a sender (a.k.a., transmitter) to a receiver.

The theory behind radio stretches back to the Persians who developed batteries by using an iron rod and copper cylinder in a clay jar with some acid material to produce low voltages as shown in **Figure 2** (you can do the same thing in a cup using a steel plate, a copper plate, and lemon juice). Unfortunately, the Persians had no use for these batteries since the electric light bulb, MP3 players, and TV would not be invented for several thousand years.

The Greeks discovered static electricity (what makes your hair stand up when you slide down a plastic board) by rubbing fur on a piece of tree resin called amber. I

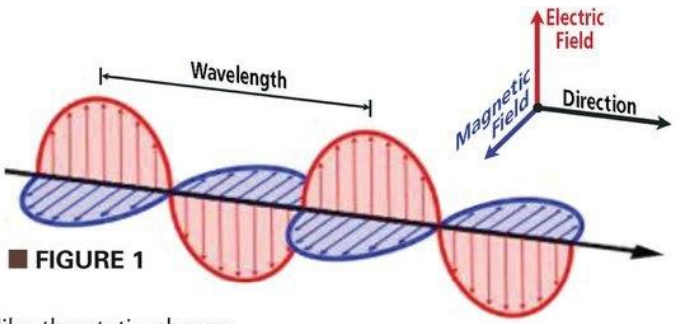


FIGURE 1

like the static charge which causes the plastic wrapper to jump back out of the trash can and attach itself to your hand.

Benjamin Franklin discovered that lightning was a moving stream of electrical particles. These discoveries were neat, but had no practical use.

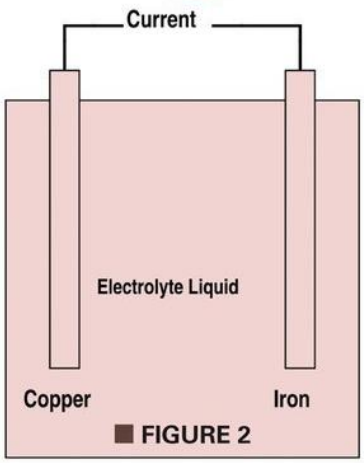


FIGURE 2

In the 1600s through the 1800s, scientists discovered many more things about electricity. Some of these discoveries were useful, and eventually lead to the development of radios. Charles Coulomb discovered like charges repel and unlike charges attract (**Figure 3**). The positive charges were atoms missing negative charges, which were named electrons (**Figure 4**). Michael Faraday discovered that if you move a magnet through a coil of wire or vice versa, you can generate a voltage (the principle of the electric generator) as shown in **Figure 5**.

Voltage pushes electrons through a wire just like pressure from a pump pushes water through a pipe (**Figure 6**). The moving stream of electrons is called current; the wire tries to keep the electrons from flowing by its resistance.

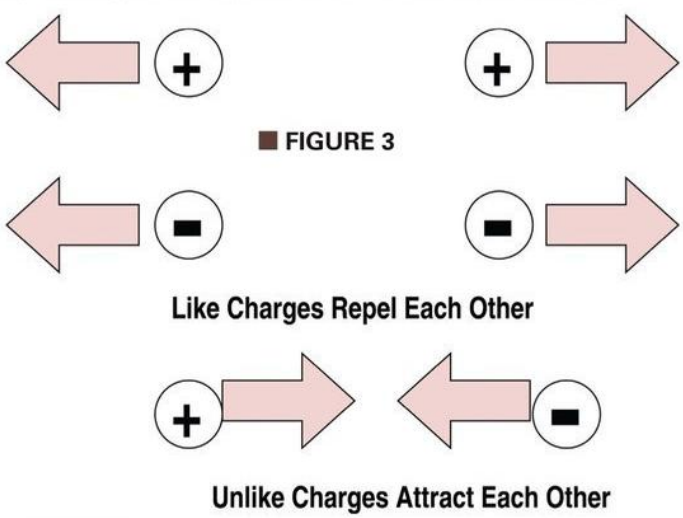


FIGURE 3

Georg Ohm (yes, this is the correct spelling) discovered that the current through a wire (I) is equal to the voltage between the ends of the wire (V) and the wire's resistance (R). As a formula, this is represented as I = V/R, or what we call Ohm's

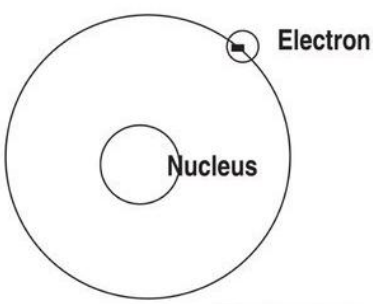
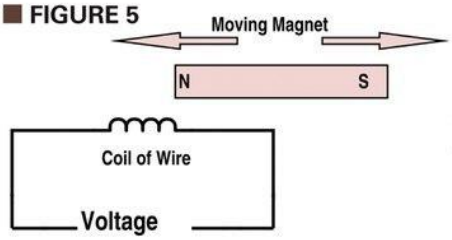


FIGURE 4
Typical Atom

Post comments on this article at
www.nutsvolts.com/index.php?/magazine/article/september2015_QA.

■ FIGURE 5



Law. Hans Christian Oersted found out that a current flowing through a wire produces a magnetic field around the wire, which will attract a compass needle (**Figure 7**).

James Maxwell later took Oersted's work that the electric field (voltage) across a wire is producing a magnetic field and combined it with Michael Faraday's generator idea (in which a magnetic field produces an electric field) to develop the theory that varying electric fields and magnetic fields could produce another series of electric and magnetic fields that can travel through the air as electromagnetic waves. We call this radio signal propagation (travel) as shown with arrows for electric and magnetic fields in **Figure 8**.

In the late 1880s, Heinrich Hertz proved that Maxwell's theory of radio propagation really worked, and in 1896, Guglielmo Marconi patented the first working radio transmitter and receiver system (amazingly, early radios were called WIRELESS because they did not need wires to carry their signals like telegraphs or telephones).

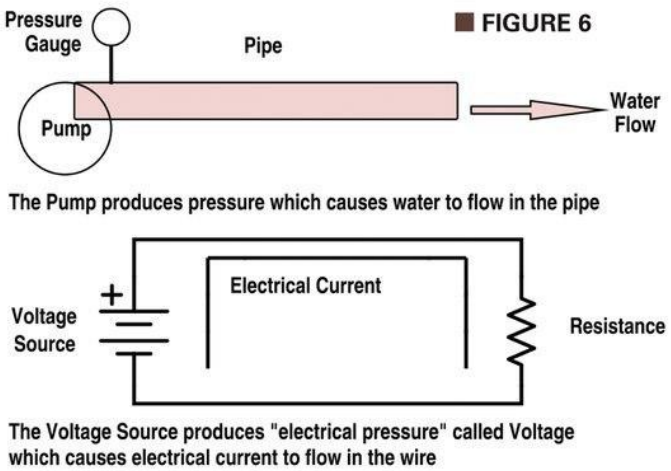
Electrical current comes in two types: Direct Current, or DC, which flows only in one direction (electrons flow from negative voltages to positive voltages); and Alternating Current, or AC, which changes directions rapidly (frequency = number of direction changes per second). The AC current is the type that comprises the radio signal.

Current (due to voltage across the wire) flows through a wire and creates a magnetic field around the wire. Let's look at current flow in a sine wave (**Figure 9** — vertical axis is current and horizontal axis is time). The current starts at zero, rises smoothly to a maximum value, drops to zero, reverses direction to drop another peak, rises to zero, and then repeats this cycle at a rate called the frequency. The alternating current produces an alternating magnetic field which, in turn, produces an alternating electric field (and so forth) for a very long distance before it becomes too weak to detect (as the distance from the transmitter's antenna doubles, radio signal strength is reduced to 1/4 of the transmitted strength).

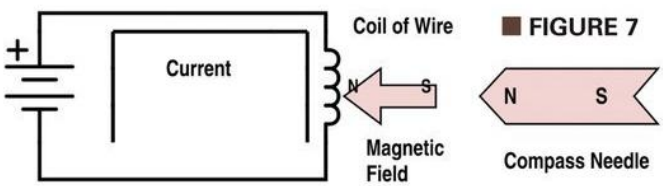
The antenna is just a length of wire that allows the current to flow and produce the electromagnetic waves like in **Figure 10**. Radio waves are the main means of communications for cellular phones, satellite, Wi-Fi interfaces, terrestrial microwave telephone transmitters, and the commercial radio you use for entertainment and news.

I have left out a lot of details regarding radio propagation theory, but I hope this simple explanation helps you to grasp the basic concepts.

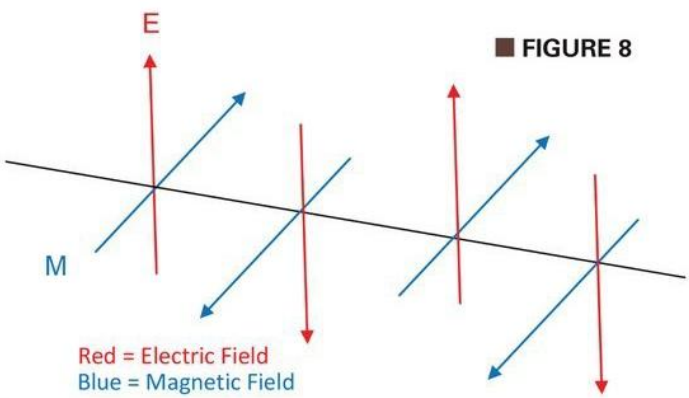
■ FIGURE 6



■ FIGURE 7



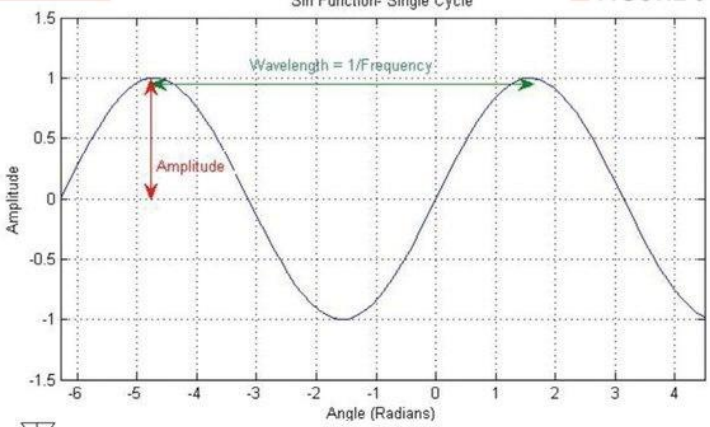
■ FIGURE 8



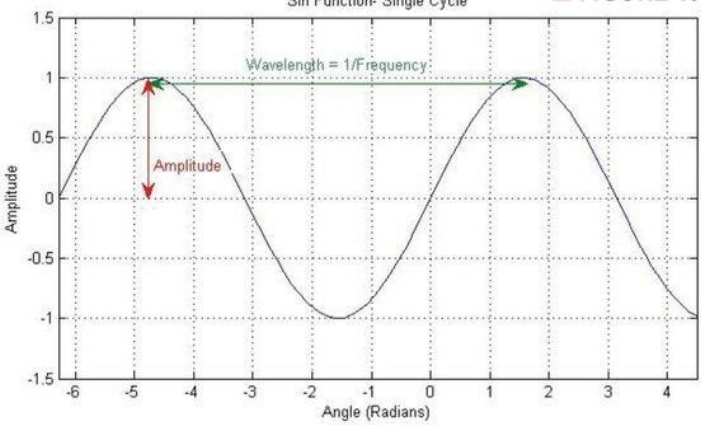
Q I am 13 years old and have recently started trying to learn electronics. Could you give me some ideas on how to approach repairing electronic equipment?

Anne Thompson
Seattle, WA

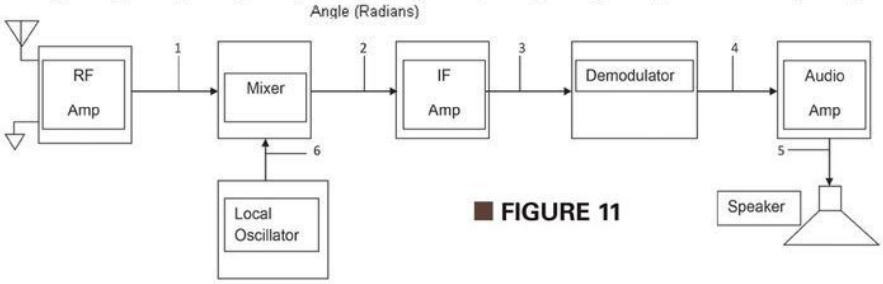
A Let me start by commending you for your interest in electronics. Electronics is both a challenging and rewarding field of endeavor. The challenging part is there are many things to learn, from circuit basics (Ohm's Law) to electronic communications to microcontroller/microprocessor programming to circuit design and layout (layout is as much an art as it is a science). The rewarding part is companies pay you good money to be able to design, operate, and maintain electronic devices, which can include the control and monitoring systems that keep



■ FIGURE 9



■ FIGURE 10



■ FIGURE 11

industry operating correctly, optimally, and safely.

Troubleshooting is the art and science of finding problems and fixing those problems so the thing you are working on functions as designed. These techniques can be applied to electronics, electrical systems, mechanical systems, and other systems (for example, medical doctors are troubleshooting when they examine you and prescribe a treatment). The "fixing" portion of troubleshooting is highly dependent on the system, and requires experience with and training on the particular system (which is too involved to be covered here).

To me, the 'finding" portion of troubleshooting is the most important and often it is the most time-consuming (once you have located the problem, the fix is usually

not that hard). Thus, we should do troubleshooting in a systematic and orderly way, such as signal injection/signal tracing or bracketing the problem. **Figure 11** shows a block diagram of a superheterodyne radio similar to the AM/FM radio you are familiar with. Each block of the diagram represents a major section of the radio's circuits, and each section is composed of electronic components such as resistors, capacitors, inductors, transistors, etc. On the diagram, I have included six test points after each of the major sections (numbered 1 to 6), with the antenna and ground symbols shown to the left of the RF amp.

Bracketing is used to narrow down the location of the problem in a complex system. This technique is based on establishing "good" brackets where the signal is as it should be, and "bad" brackets where the signal is either absent or degraded.

Referring back to **Figure 11**, we can assume the antenna and ground are good by making sure it is attached properly (you can use a known good radio to ensure there is a radio signal present, or use an RF signal generator to send an input signal) to establish a good bracket. (By convention, signals on diagrams flow from left to right.)

Let's say that there is no signal coming from the speaker, so this represents our bad bracket. We can either move to a good bracket to the right, move the bad bracket to the left, or both. For this case, let's move the bad bracket until we find a good signal. This way, we'll know that the problem is between the last bad bracket and the last good bracket. At this stage, we use a detailed schematic diagram to determine which component(s) are causing the problem.

Signal insertion/signal tracing is used in conjunction with a signal generator and a signal detector (DMM, VOM, oscilloscope, logic probe, etc.). Again, in **Figure 11**, you could insert a radio frequency signal into the antenna connectors, then use an oscilloscope to see if there is an audio signal at the speaker terminals and demodulator output at the intermediate frequency (455 kHz for most broadcast band sets), then at the radio frequency signal (535 to 1,700 kHz for AM and 88 to 108 MHz for FM) at the RF amp output and local oscillator frequency (carrier frequency + 455 kHz).

You can see there is a broad range of frequency covered in a "simple" AM/FM radio. The signal input to a TV set is a visual pattern that appears on the screen of a good TV. Seeing actual broadcast signals is difficult since the broadcast signal pattern is not as periodic as that of a signal generator.

Bracketing and signal injection/signal tracing can be used on systems other than electronics, such as wastewater treatment systems and mechanical systems. However, the "test points" may not be as easily determined as they are in electrical systems.

I hope this information whets your appetite for further study into electronics to help you in your troubleshooting efforts. **NV**

■ BY RON HACKETT
www.jrhackett.net

PICAXE-PC Serial Communication — Part 3

In Part 2 of our PICAXE-PC serial communication project, we moved up to the 20X2 processor and experimented with one way of speeding up the response time for incoming serial data that has been received in the background. At the end of that article, I mentioned there's an interrupt-based method of improving the 20X2 response time, so that it's even faster than our previous approach. This month, we'll experiment with that approach, but first we need to discuss some of the details of the commands and built-in variables that we'll be using. Let's get started.

As we've already discussed, only the X1-class and X2-class processors contain a scratchpad memory area: the 28X1, 40X1, and 20X2 processors contain 128 bytes of scratchpad memory (numbered 0-127); and the 28X2 and 40X2 processors contain 1024 bytes of scratchpad memory (numbered 0-1023). We can use the scratchpad memory area for the temporary storage of data in any program, and it's especially useful for arrays.

However, if we configure a program to use the hardware serial input (*hserin*) pin, the processor automatically uses the scratchpad to store the serial data that's received in the background. As a result, we would need to be careful if we wanted to store temporary data in the scratchpad and also use it for hardware serial input.

For example, we could decide to use the first 64 locations in the 20X2 scratchpad (0-63) for serial input, and the remaining 64 locations (64-127) for temporary storage of other data, but we would need to be sure that we never receive more than 64 bytes of data. If that happened, the incoming bytes would over-write our stored data, and the program would behave erratically.

Scratchpad-Related Commands

PICAXE BASIC includes two scratchpad-related commands: *get* and *put*. The *get* command is used to read data from the scratchpad. (In other words, the *get* command "gets" data from the scratchpad.) The full syntax is *get location, variable, variable, WORD wordvariable ...* where *variable is* the name of a byte variable that receives the data, and *WORD wordvariable* refers to the name of a word variable. (To use a word variable, the keyword *WORD* must be used before the *wordvariable* name.)

We used a simple form of the *get* command in the previous Primer column, but this month I want to include a little more detail. The following example statements should clarify the syntax. In each example, *loc* refers to a location in the scratchpad:

```
get 10, b1
' value at loc 10 is copied to b1

get 0, b1, b2, b3
' value at loc 0 is copied to b1
' value at loc 1 is copied to b2
' value at loc 2 is copied to b3

get 5, b0, word w3
' value at loc 5 is copied to b0
' value at loc 6 is copied to low-byte of w3
' value at loc 7 is copied to high-byte of w3
```

The *put* command is used to write data to the scratchpad. (In other words, the *put* command "puts" data into the scratchpad.) The full syntax is *put location, variable, variable, WORD wordvariable,* where *variable* is the name of a byte variable whose value is being written to the scratchpad, and *WORD wordvariable* is the name of a word variable. (Again, to use a word variable, the keyword *WORD* must be placed before the *wordvariable* name.)

The following example code statements should clarify the syntax. Also, *loc* again refers to a location in the scratchpad:

```
put 10, b1
' value of b1 is copied to loc 10

put 0, b1, b2, b3, b4
' value of b1 is copied to loc 0
' value of b2 is copied to loc 1
' value of b3 is copied to loc 2
' value of b4 is copied to loc 3

put 5, b0, word w3, b1
' value of b0 is copied to loc 5
' low-byte of w3 is copied to loc 6
' high-byte of w3 is copied to loc 7
' value of b1 is copied to loc 8
```

Scratchpad-Related Special Function Variables

There are four special function variables that are built into the PICAXE compiler for use with the scratchpad memory area:

ptr – the scratchpad pointer
@ptr – the value pointed to by *ptr*
@ptrinc – the value pointed to by *ptr* (post increment)
@ptrdec – the value pointed to by *ptr* (post decrement)

Since the brief descriptions above are a little cryptic (to say the least), let's take a look at each one in more detail. We've already used the *ptr* (pronounced "pointer") variable in the previous installment of the Primer, but it's worth mentioning it again. The *ptr* variable contains the number of the location that we want to access.

For example, suppose the value 65 is currently stored in scratchpad location 3. If we wanted to retrieve that value, we could write the following code snippet (assuming the *char* variable has already been declared in a symbol statement):

```
ptr = 3
get ptr, char
```

When the above code snippet is executed, the value of 65 would be stored in the *char* variable.

In order to understand how the *@ptr* (pronounced "at pointer") variable functions, we need to briefly mention the distinction between direct addressing and indirect addressing, which is another topic we've discussed in the past but it's also worth explaining again.

The *get* statement in the above code snippet is an example of direct addressing because we're directly accessing the value stored in location 3 of the scratchpad. Indirect addressing, on the other hand, uses the concept of a "pointer" variable, which "points to" the data we want to access. In other words, we indirectly access the data we want. For example, again assume that ptr = 3, and 65 is currently stored in scratchpad location 3. Now, consider the following two code statements:

```
char = ptr
char = @ptr
```

In the first statement, we're using direct addressing to access the value of *ptr* (which is 3), and store it in the *char* variable. In the second statement, we're indirectly accessing the data byte (65) that's stored at scratchpad location 3. In other words, *char* receives the value (65) that's pointed to by *ptr*.

As you can see, indirect addressing is more complicated than direct addressing. However, it's also faster than direct addressing which can be very helpful when processing serial data that arrives in the background.

Now, let's discuss the next two special function variables: *@ptrinc* (pronounced "at pointer inc") and *@ptrdec* (pronounced "at pointer dec"). The *@ptrinc* variable accomplishes two tasks. First, just like *@ptr*, it points to the value stored in location *ptr* of the scratchpad. Second, whenever *@ptrinc* is used in a program statement, the value of *ptr* is automatically incremented **after** the statement is executed (which is why *@ptrinc* is often referred to as a post incrementing instruction).

For example, suppose the first three locations in the scratchpad (*locs* 0-2) contain the values 65, 66, and 67 in that order. Now, consider the following code snippet:

```
ptr = 0
b1  = @ptrinc
b2  = @ptrinc
b3  = @ptrinc
```

After the above snippet has executed, b1 = 65, b2 = 66, b3 = 67, and ptr = 3. The *@ptrdec* variable functions in the same manner as *@ptrinc*, except that the *ptr* variable is automatically decremented after each execution of a statement that includes *@ptrdec*.

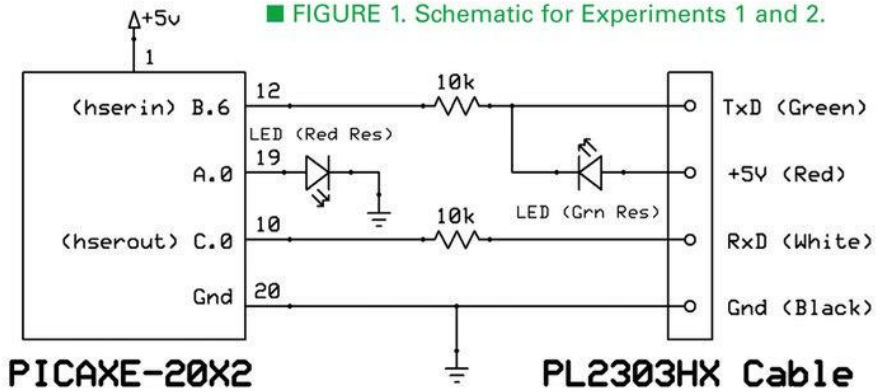
There are also two system variables that we'll be using again this month: *hserptr* (a.k.a., *hserinptr*) and *hserflag* (a.k.a., *hserinflag*). We don't need to discuss them any further, but I do want to repeat that I prefer the names *hserinptr* and *hserinflag* because I think they are more descriptive. Of course, if you prefer the shorter names, feel free to edit this month's programs.

At the end of the previous Primer column, I indicated that this month we're going to use the *setintflags* command to develop an interrupt-based solution to our task of speeding up the response time to hardware serial input. I suggested that you might want to read through the documentation. However, I do realize that not everyone enjoys wading through documentation, so let me briefly summarize what's relevant for our current project.

Except for the *hserinptr* and *hserinflag* variables, everything we've discussed so far this month is limited to the X1- and X2-class processors. Unfortunately, that's also the case for *setintflags*: M2-class processors do not include a built-in mechanism for interrupting a program whenever background serial data is received. The *setintflags* command has five different syntax variations. Fortunately, we only need to use the following two:

```
setintflags flags, mask
setintflags OFF
```

The second variation is self-explanatory, so let's focus on the first one. In the previous column, I mentioned that the *hserinflag* variable is one of the bits in the *flags* system variable. I didn't specify which one because we didn't need the information at that time. This month, however, we do need to know that *hserinflag* is bit 5 of the *flags*



■ FIGURE 1. Schematic for Experiments 1 and 2.

variable. In the first syntax variation above, the *mask* parameter of the *setintflags* command can be used to specify which *flag* bit(s) we want to trigger the interrupt; any *flag* bit masked by a "0" will be ignored. The *setintflags* command allows for various combinations, but we're only interested in the *hserinflag* (bit 5 of *flags*), so we'll specify *mask* as %00100000, and ignore all the other flags.

The *flags* parameter can be used to specify which condition (high or low) of the *flag(s)* we want to trigger an interrupt. Again, we're only interested in the *hserinflag* which is automatically set to "1" (high) when background serial data is received. So, we will also specify *flags* as %00100000. Actually, because we're ignoring all the other flags, it doesn't matter what value (0 or 1) we place in any of the other bit positions, but it could be confusing to change any of the other zeros to ones.

By now, I'm sure you're glad to hear that we've finally covered all the theoretical details that we need for our first experiment! This would be a good time to take a break and maybe stretch a little. When you're ready, we'll move on to our first experiment.

Experiment 1: *Hserin* with an Interrupt and Direct Addressing

This experiment is very similar to our two 20X2 experiments from last time. The main difference is that we will be using the *hserinflag* interrupt this time. **Figure 1** presents the schematic for both experiments this month, and **Figure 2** is a photo of my breadboard setup. As we did previously, we'll again be using the CoolTerm program (or whichever terminal program you prefer) in line mode, so that we can type a complete line of text and then press the enter (or return) key on the Mac or PC to transmit the entire line of serial data to the 20X2.

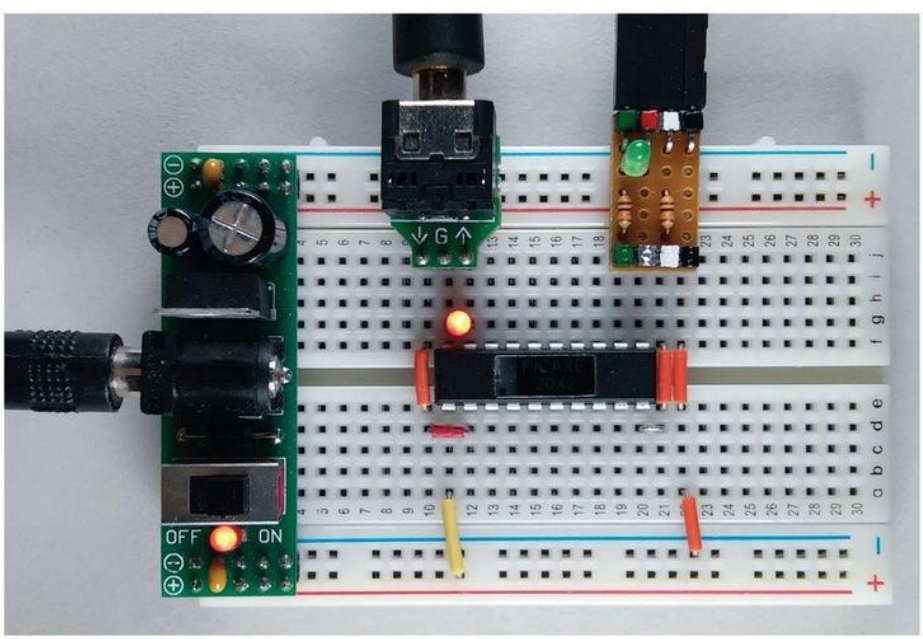
I also again used 115,200 for the baud rate, but if you had any problems with that previously you may want to drop back to 57,600 baud. (We'll discuss baud rate changes shortly.) When you're ready to configure your terminal program for the experiment, be sure to specify the following settings:

- In the "Serial Port Options," select 115,200 baud and whichever serial port your PL2303HX cable is connected to. (Mine always shows up as "usbserial;" the same may be true for your cable.)
- In the "Terminal Options," select "Line Mode" and enable the "CR+LF" option for the "Enter Key Emulation."

Previously, we used the final LF character (ASCII 10) in each data transmission to determine when to stop processing the serial input data. This time — as we will soon see — we're going to use a different approach, but we still want the CR and LF characters to display the data on separate lines in the terminal program.

The software for Experiment 1 (***Hser20X2IntDirect.bas***) is available at the article link. Now would be a good time to get it, along with the other program we'll be using this month. Before you run the program on your 20X2 breadboard setup, let's take a look at the details of the interrupt subroutine. (You may want to print a hard copy of the program as a reference for our discussion.)

As usual, the following numbered



■ FIGURE 2. Breadboard setup for Experiments 1 and 2.

comments refer to the corresponding numbers along the right edge of the program listing:

[1] We're using the same *hsersetup* command we used previously. The *mode* parameter is configured as follows: background receive to scratchpad, true serial input and output, *hserin* and *hserout* pins enabled.

[2] We've already discussed the *setintflags* command that we'll be using this month, so there's no need to elaborate further.

[3] If you have used the *setint* command in any of your projects, you know that the first statement in the interrupt subroutine is usually *setint off*. The purpose of that statement is to make sure the interrupt subroutine itself will not be interrupted while it's being executed. If that were to happen, the program would behave erratically.

In the current program, it's highly unlikely that additional background serial data would be received while the current data is being processed, but including the *setintflags off* statement protects the program from that remote possibility. Of course, if a second microprocessor were transmitting the data, you would need to make sure to include a brief pause between data transmissions.

[4] At this point, it's important to remember that the interrupt is triggered as soon as the first data byte is received in the background. Even at a rate of 115,200 baud, a string of data that's possibly as long as 128 characters will still be in the process of being received at the beginning of the interrupt. Therefore, a program delay is needed at this point to make sure all the data has arrived before we begin to process it.

In order to determine how long the delay should be, we need to do a little approximate arithmetic.

- Each incoming byte contains one bit: 1 start + 8 data + 1 stop
- Therefore: 128 bytes = 1280 bits
- Form a proportion: 115,200 bits / 1 sec = 1280 bits / x sec
- Therefore: 115,200 * x = 1280 * 1
- So, x = 0.011 seconds, or 11 mS

(I decided to increase the delay to 15 mS, just to be safe.) If you want to run the program using a different baud rate, the necessary delay will need to be recalculated. To save you the trouble, **Figure 3** gives the necessary delay for selected alternate baud rates.

hsersetup Baud Rate	Interrupt Pause (mS)
4800	360
9600	180
19200	90
38400	45
57600	30
115200	15

■ FIGURE 3. Interrupt pause needed for various baud rates.

[5] When the program first begins to run, *hserinptr* is automatically initialized to 0 (as are all PICAXE variables). Therefore, when the interrupt occurs, the first received byte is stored at location 0 in the scratchpad, and *hserinptr* is automatically incremented to 1 in preparation for the next byte. This process continues for every byte that's received in the background.

As a result, when all the data has been received, *hserinptr* points to the location in the scratchpad that is one greater than that of the last received character. Therefore, at this point, we decrement *hserinptr* by 1, so that it points to the location of the last received byte.

[6] Here, we loop through the scratchpad locations from location 0 to the location of the last received byte using direct addressing to get each character, and then echo it back to the terminal program.

[7] Before returning from the interrupt subroutine, we need to reset *hserinptr* to 0, so that storage of the next serial input data will again begin at location 0 in the scratchpad.

[8] We also need to reset the *hserinflag* to 0 in preparation for the next background serial transmission.

[9] The very last thing we need to do before returning from the interrupt subroutine is to re-enable the interrupt so that the next background serial transmission will again trigger it.

At this point, we're ready to test the ***Hser20X2IntDirect.bas*** program, but before we do that, I want to correct a mistake I made in the *hserial20X2.bas* program we used in the previous Primer. In that program, I included a note stating that pin A.0 is "undocumented" on the 20X2 (as it is on the 20M2). However, that's not the case. If you refer to the 20X2 pinout, you will see that pin A.0 is, in fact, documented. In the present program, I corrected that error.

Before running the ***Hser20X2IntDirect.bas*** program, make sure your terminal software is set up as described previously, and that your breadboard circuit is set up with a 20X2 processor in place (see **Figure 2**).

When you're ready, download the program to your breadboard setup and test several different input strings. An exact duplicate of each string should be echoed back to the terminal. If not, you will need to check your breadboard setup for wiring errors.

Experiment 2: *Hserin* with an Interrupt and Indirect Addressing

This experiment essentially replicates Experiment 1, except this time we're using indirect addressing to echo the data back to the terminal. In Experiment 1, the

following *for/next* loop accomplished this task:

```
for ptr = 0 to hserinptr
  get ptr, char
  hserout 0, (char)
next ptr
```

In Experiment 2, we're going to replace the *get* statement (direct addressing) and the *hserout* statement for the actual echoing of data with a single indirect addressing *hserout* command:

```
for ptr = 0 to hserinptr
  hserout 0, (@ptr)
next ptr
```

That's the only change we need to make. (Of course, in the indirect version, we no longer need the *char* variable.) You can either edit the interrupt subroutine to make the necessary changes and save the new file as Hser20X2IntIndirect.bas, or simply use the file that's provided at the article link.

When you download and run the program on your 20X2 breadboard setup, you probably won't even notice the increase in speed because everything happens so quickly anyway. However, the elimination of even one statement in a loop that can execute as many as 128 times does significantly speed things up.

Before we move on, there are a few details that are worth mentioning. First, try a little experiment. Type a data string into your terminal program, but before pressing the return key on your computer, watch the blinking of the LED on the breadboard. Time your key-press so that it coincides with the start of one of the "on" portions of the blink. What happens, and how do you explain that?

You should have observed that the "on" portion of the blink was almost immediately terminated, rather than lasting for the expected two second delay in the main program loop. That happens because an interrupt subroutine behaves the same way as any subroutine: The compiler notes which program statement was interrupted, and when the subroutine has finished executing, the compiler resumes program execution at the program statement *after* the one that was interrupted.

In other words, when you press the return key and interrupt the program near the beginning of the *wait 2* statement, that statement never finishes timing out. Program execution continues at the *loop* statement, so the LED turns off almost immediately. Of course, you may have a *wait* statement in a program that shouldn't be truncated by an interrupt. If so, you would need to replace it with a loop that divides it into many shorter delays. For example:

```
for index = 1 to 200
  pause 10
next index
```

That way, program timing would only be off by 10 mS, rather than two seconds.

Second, I should mention the importance of avoiding "rollover" in the scratchpad. As you know, the 20X2 scratchpad contains 128 bytes of memory at locations 0 through 127. In order to function correctly with the scratchpad, the *ptr* and *hserinptr* variables are different in one important way from all other PICAXE variables. Of course, all byte variables can have a maximum value of 255: Whenever a program statement tries to increase that value, the byte variable "rolls over."

For example, if the b0 variable = 255 and we add 1 to it, it rolls over to 0. (Also, if the b0 variable = 255 and we add 5 to it, the variable rolls over to 4.) On the other hand, word variables can have a maximum value of 65,535. If we try to increase their value beyond that, they also roll over.

In order to work properly with the 128-byte scratchpad, the *ptr* and *hserinptr* variables both roll over above 127 on the 20X2 processor. As a result, if we send a string of more than 128 serial bytes to the 20X2, we will lose the first 128 bytes! For example, if we send a string of 132 bytes, *hserinptr* will both end up containing the value 4. As a result, when we echo the data, only the last four characters (in scratchpad, locations 0 through 3) will be sent back to the terminal. Don't take my word for it — try the following experiment:

Type the full lower case alphabet into a text editor or word processor. Copy the entire 26 characters and paste them to the end of the text string four times. (The text should contain 5 * 26 = 130 characters.)

Copy the entire 130-character string, paste it into the transmit box in your terminal, and press the return key. (Don't forget, the terminal will append the CR and LF characters, so the transmitted data will contain a total of 132 characters.)

You should see "yz" echoed in the terminal. The CR and LF characters were also echoed, so that's a total of four characters; the first 128 characters were lost! The moral of the story is, if you're using another processor to send data to the 20X2, make sure that no data string exceeds 128 characters.

Finally, if you compare this month's experiments to those of the previous installment, you can see this month we used a different approach to determining the last data byte in the serial string. Previously, we used the final line feed (LF) character in each data transmission to determine when to stop processing the serial input data.

This month, however, we used *hserinptr* for that purpose. That may seem like a minor point, but this month's approach does have a major advantage. If we want to transmit numerical data from a remote processor, our earlier approach would preclude the possibility of transmitting a data value of 0 (unless we converted everything to ASCII characters) because a value of 0

would stop the processing of the data. Using *hserinptr* removes that limitation and allows us to transmit any data value — including 0.

What's Next?

At this point, we've covered the basics of receiving serial input in the background while our program is executing other tasks. Hardware serial input is a powerful technique that's available on all current M2- and X2-class PICAXE processors. So far, we've been experimenting with its capabilities by manually sending data strings from a terminal program running on a PC, but now we're ready to move on to using a "master program" on the PC to communicate with a PICAXE project.

As I mentioned in the first installment of our PICAXE-PC serial communication project, Python is an ideal programming language for this purpose. We've already covered the basics of Python programming in our PICAXE-Pi projects, and Python is freely available for all three major PC operating systems. In fact, some version of Python comes pre-installed on OS X and most Linux distributions. Windows users, however, will need to download and install Python on their PC.

Even if you already have Python installed, you may want to update to the latest version. It's probably also a good idea to install the latest releases of both Python2 and Python3 — especially if you haven't yet decided which version you want to use. A quick search for "install Python" will locate many resources, but I can also recommend either of the following two sites:

- The "official" python site (**python.org**): Click on the "Downloads" tab, and then choose the latest Python releases for your operating system. Also, see the Beginner's Guide under the "Documentation" tab.
- "The Hitchhiker's Guide to Python!" (**docs.python-guide.org**): This site contains good information on downloading and installing Python, as well as guidelines for writing Python code.

In the next Primer installment, we'll experiment with a couple of simple Python "master programs" that will request data from a PICAXE project, and update the data in real time on the PC. Before then, you may want to make sure you have the latest versions of Python installed on your PC, so that we can hit the ground running!

In the meantime, have fun ... **NV**

Getting Back into Ham Radio

Trials And Tribulations

I have been a ham for many decades, but I have not been active these past few years. I started out with a novice license as WN5TOM. Later, I upgraded to general class as W5TOM and then again as K3CTX. Ultimately, I passed the Extra exam. I was W8LJR while in Michigan at Heathkit. Back home in Texas, I became W5LEF.

What has really kept me away from the hobby are the antenna restrictions in my subdivision. No antennas are allowed except for small satellite dishes from DirectTV or DISH. So far, I have not figured out how to hide the kind of wire antenna I want and need. It occurred to me that some mobile or portable unit could get me back quickly. So, I acquired a handheld – specifically, a Kenwood TH-F6A (see **Figure 1**). It is about as big as a deck of cards but contains radios for the two meter (144-148 MHz), 1.25 meter (222-225 MHz), and 70 cm (420-450 MHz) VHF/UHF bands. The main operating mode is FM and the maximum output power is five watts. Not bad for something so small. Here is my experience with this exercise.

Operating Difficulties

I fondly remember the days when ham radios just had a big tuning dial and a few knobs and switches. Not so today. Virtually every radio these days is microcomputer controlled, so comes with a keyboard, a slew of buttons, and an LCD screen. Using these keys and multiple menus, you set the operating frequency and other parameters. This is no easy task. A 58 page manual helps you figure out how to set up the radio. It took me a couple days of playing around with the unit before I could get it working.

The whole process is cumbersome and off-putting. It takes multiple button push sequences to do the simplest of operations. It is a giant pain. Not only that, the buttons are tiny and close together so there are entry errors if you have fat fingers. In some cases, I had to use a magnifying glass to see the display.

There are some benefits to having a small handheld transceiver (HT), but a few downsides as well. I can't blame the manufacturer (Kenwood) as all other HTs are the same way. I would much rather have a larger, easier to use unit. Call me old-fashioned.

Range Limitations

Operating at VHF or UHF frequencies is great since the antennas are small. However, transmission is line of sight (LOS), meaning antenna to antenna. In other words, the transmit antenna needs to "see" the receive antenna. That means antennas must be as high as possible to get any results.

Radio waves do pass through

■ **FIGURE 1.** The Kenwood TH-F6A is a tri-band handheld FM transceiver for the 144, 222, and 440 MHz ham bands. Lots of buttons and menus make it a challenge to use.

Post comments on this article and find any associated files and/or downloads at
www.nutsvolts.com/index.php?/magazine/article/September2015_OpenCommunication.

many objects but are greatly attenuated. Furthermore, physics state that higher frequencies travel over shorter distances for any given power and receiver sensitivity. What all this boils down to is a very short potential range. For a handheld, the range is probably only a few miles at best unless you stand on a tall building or put up an external antenna – which I cannot do.

Anyway, those facts should have dawned on me when I started this process. I live in a small town outside of Austin, TX. Most of the ham activity is within the city, so my potential for making contacts is minimal. Nevertheless, I tried.

Checking Out the Two-Meter Band

The two-meter band has more activity and longer range potential, so I focused my efforts here. One benefit of this band is the availability of repeaters to extend the range. A repeater is a station that is usually located at a high point so it can cover a wide area. The repeater receives a signal from a low power handheld or mobile unit, then retransmits it on a different frequency at a higher power level. A cell phone tower site does the same thing.

For example, in the two-meter band, a handheld will transmit on 146.34 MHz – the receive frequency of the repeater. The repeater retransmits on 146.94 MHz. The 0.6 MHz spacing allows the repeater to utilize the same antenna for transmit and receive using a duplexer – a filter unit that keeps the simultaneous signals separate. The repeater produces a huge increase in the coverage range of portable units.

I checked the American Radio Relay League (ARRL) repeater directory to locate potential nearby repeaters. I identified about a dozen potential candidates. Using the scan function on the HT, I was able to pinpoint some I could hear. Unfortunately, I was too far away to make contact.

Another factor is that most repeaters use a continuous tone coded squelch system (CTCSS) to access a repeater. This is the use of a unique audio tone that a transmitter must momentarily transmit before the repeater will respond. There are 42 possible tones from 67 Hz to 254.1 Hz. You need to know that tone to use the repeater. Most HTs can be set up to automatically transmit the tone once you know it. The ARRL repeater directory gives these tones as well as the operating frequencies.

The only way that I was able to make any contacts was to drive into Austin at some elevated point where I could access several repeaters. I made many contacts to confirm the HT's operation, which was satisfactory. Unfortunately, most repeater users do not seem to engage in casual rag chewing, as in using other bands and modes. Many repeaters belong to clubs so the users are generally familiar with one another.

Back in the Saddle

So, I am back on the air. Sadly, it's been a somewhat disappointing experience. The combination of an agonizingly hard-to-use transceiver and the short range really limits what you can do. Unless you use repeaters or put up a high outside antenna, potential contacts will be few and far between. HTs are probably great for emergencies, but are not so enjoyable as a ham experience. (Just my opinion, of course.)

My next attempt will be to get back on the high frequency (HF) ham bands using CW. That will occur when I can figure out what to do about a stealth antenna. I'll let you know how that goes in a future column. **NV**

NEW PRODUCTS

- HARDWARE
- SOFTWARE
- GADGETS
- TOOLS

HIGH SPEED USB
DIGITAL I/O DEVICE

Measurement Computing Corporation announces the release of the USB-DIO32HS high speed digital I/O device. The USB-DIO32HS features 32 high speed digital I/O channels that are 50 times faster than previous devices from the company.

The USB-DIO32HS features input and output rates up to 8 MS/s. Independent input and output scan clocks, hardware and software triggering, and pattern detection/generation are also included.

Microsoft Windows software options for the USB-DIO32HS include TracerDAQ to display and log data, along with comprehensive support for C, C++, C#, Visual Basic, and Visual Basic .NET. Support is also included for DASYLab and NI LabVIEW.

Support for Android devices allows users to develop DAQ applications for tablets and smartphones. Sample applications are available for free download on Google Play. Retail price is $499.

For more information, contact:
Measurement Computing
www.mccdaq.com

HALF-WAVE ANTENNAS
EXPAND TO 2.4 GHz

Linx Technologies announces the addition of 2.4 GHz to its existing permanent-mount PML series of half-wave dipole antennas. This new design adds more options for antenna design in the 2.4 GHz ISM band.

With multiple standards operating at 2.4 GHz — such as Bluetooth, Thread, Wi-Fi, ZigBee, and others — the vast growth of the Internet of Things (IoT) is driving the added need for more options to antenna design. Where permanent mount design is needed, the PML series half-wave whip antenna has outstanding performance in a rugged and cost-effective package.

Featuring a flexible shaft that protects the antenna from shock and harsh weather conditions, the antenna is attached by placing its base through a 1/4" (6.35 mm) hole in the product and securing it with a nut or by threading it into a PEM-style insert.

Also available with an optional 1.32 mm coax cable terminated with a U.FL/MHF compatible connector, this increase of product line options saves the labor of adding a connector while using one that is small enough to fit through the antenna's mounting hole. Custom lengths and terminations are available by special order.

The 2.4 GHz frequency band is universally accepted worldwide and is shared among many applications and standards: IEEE 802.15.4, Bluetooth, IEEE 802.11/Wi-Fi, ZigBee, Thread, and other personal area networks. All of the devices require some kind of effective antenna design, and this design helps in many IoT applications where a permanent mount, high-performance half-wave antenna is needed.

For more information, contact:
Linx Technologies
www.linxtechnologies.com

PROBE
ADAPTERS

Teledyne LeCroy has introduced two new products that provide a simple and easy interface of third-party probes and current measurement devices to Teledyne LeCroy oscilloscopes. The first — the TPA10 TekProbe Probe Adapter — adapts a wide variety of Tektronix voltage and current probes; the second — the CA10 Current Sensor Adapter — adapts a wide variety of third-party current measurement devices.

Both adapters attach to the Teledyne LeCroy ProBus probe interface — a standard probe interface that for the past 20+ years has been present on the vast majority of oscilloscopes shipped by Teledyne LeCroy. These adapters will help users better leverage their existing Tektronix probe and third-party current measurement device investments with Teledyne LeCroy oscilloscopes.

The TPA10 TekProbe permits connection of select Tektronix TekProbe interface level II probes to any ProBus-equipped Teledyne LeCroy oscilloscope. The TPA10 automatically detects which a Tektronix probe is attached; it supplies all necessary power and offset control to the probe, and communicates the probe signal to the Teledyne LeCroy oscilloscope.

Supported probes include many popular Tektronix probes, preamplifiers, current probes, single-ended active probes, and differential active probes, such as Tektronix's ADA400A differential preamplifier, their TCP202 and TCP202A current probes, the P6205, P6243, P6245, P6241, and P6249 single-ended active probes, and their P6246, P6247, P6248, P6250, P6250, P5205, P5205A, P5210, and P5210A differential and high voltage differential active probes.

The CA10 current sensor adapter provides the ability for a third-party current measurement device to operate like a Teledyne LeCroy probe. The CA10 is programmable and customizable to work with third-party current measurement devices that output voltage or current signals proportional to measured current.

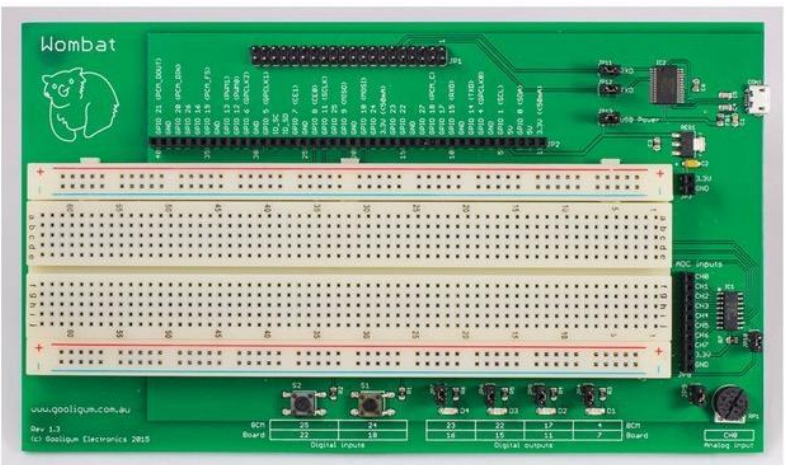
A simple interface provides a user with the ability to program the CA10 to contain the specifications of the current measurement device so as to automatically correct for the gain or attenuation, and display results in ampere units. The CA10 also provides the ability to easily install physical hardware components such as shunt resistors and bandwidth filter components based on the requirements of the device being used.

Once the CA10 is programmed and configured, the

current transducer/transformer will be recognized when attached to any compatible Teledyne LeCroy oscilloscope or motor drive analyzer. This saves time and possible errors involved in manually entering scaling factors and units each time the device is connected. Examples of devices that can be used with the CA10 include Pearson current transformers, Danisense current transducers, PEM-UK Rogowksi coils, or any conventional turns-ratio current transformer.

The TPA10 is priced at $950; the CA10 is priced at $295. There is also a QuadPak available (qty 4 of each device) for $3800 and $1180, respectively. The QuadPak includes a soft-carrying case to store the adapters. Delivery time for each item is 4-6 weeks.

For more information, contact:
Teledyne LeCroy
www.teledynelecroy.com



WOMBAT
PROTOTYPING BOARD

Anyone interested in creating their own Raspberry Pi projects should take a look at the Wombat prototyping board, recently released by Gooligum Electronics. It's designed to make it easy to prototype original circuits that connect to a Raspberry Pi – whether playing with ideas or designing a new product – freeing up time from the nitty gritty so the more challenging task of seeing a design come to fruition can be focused on.

The newer Raspberry Pi models provide 40 GPIO pins which is ideal for experimenting and product development. However, connecting those pins to circuits can be messy and error-prone. The Wombat simplifies access to the Pi's GPIO pins by breaking them out to a clearly labelled header alongside a large solderless breadboard (mounted on a sturdy base supported by rubber feet), which won't limit users to building small projects.

Although the Pi's GPIO pins operate at 3.3V, its onboard 3.3V regulator is not intended to power external devices, so the Wombat includes a 3.3V regulator which can supply up to 500 mA – enough to power more complex projects.

The Wombat also provides eight analog input pins (something the Raspberry Pi lacks) for use with analog sensors, along with a trimpot that can be deployed as a simple analog control. It includes a number of LEDs and pushbutton switches for use as digital outputs and inputs, or for testing.

The analog inputs along with the LEDs and pushbuttons are all made available through a Python module. The functions are fully documented and clear examples are provided, making it easy to adapt them to a project, using them in Python 2 or 3 programs. For those not using Python, the comprehensive documentation includes full schematics.

The Wombat adds a USB serial console to the Raspberry Pi via a genuine FTDI serial-to-USB bridge, making it possible to set up a Pi or debug problems when the Pi is running headless and the network connection isn't set up yet – such as when taking a Pi to a makerspace or jam. The Wombat's micro USB connector can also be used instead of the Pi's power connector to power both the project and the Pi. The Pi and project can be powered and controlled all through a single cable (as long as the laptop's USB port supplies enough current; many can), making for a very clean uncluttered setup.

A set of introductory projects featuring an RGB LED and light and temperature sensors is bundled with each board, including all the project components and a set of breadboard jumpers. Each project is fully documented with source code provided.

The Wombat board retails for $50. Bulk purchase and educational discounts are available.

For more information, contact:
Gooligum Electronics
www.gooligum.com

Continued on page 79

If you have a new product that you would like us to run in our New Products section, please email a short description (300-500 words) and a photo of your product to:

newproducts@nutsvolts.com

By Bill Elwell

Little Book of Horror

Browsing in a craft store, I happened on a box that looked like a small book. I thought it would be cool to see if I could put enough electronics in it to make the cover pop up as a Halloween prop. As with most of my projects, that concept was just the beginning of a small project that became a larger project due to scope creep. I would always think — wouldn't it be even cooler if I did this ... and then that!

This is really a fun project. Even though it took six prototypes, I enjoyed every minute of the design/construction process.

With all the Halloween props on the market, it's difficult to startle people. I use the word *startle* because 'scare' is now a multi-million dollar Hollywood thing. In this project, you can build a little book that startles people.

A good Halloween prop needs a "hook" to pull people in; a visual to keep them interested; then a "shock" to startle them. This project **has all three**: a hook (muffled sound) pulls the victim in; a visual (an eye moving left to right) keeps things interesting; and then there's a make-you-jump finale — all in a small 5 by 7 inch book. I have tested the effectiveness of this little book in my wife's gift shop (I didn't scare any little old ladies — I swear!). Although I didn't video any of the reactions, they were video-quality startles.

What Exactly Does It Do?

It's a small, odd, old-looking book — with an eye and eyebrow. It sits quietly on a surface. It then makes a weird muffled laughing sound. The victim hears the sound and walks up to investigate. Nothing happens. No, wait! Did that eye move? They look closer, reaching out to open the cover of the book — Yikes!

It just jumped at the unsuspecting mark several times, opening and closing its "mouth." The book then retreats to assume its passive-aggressive demeanor — for the next victim.

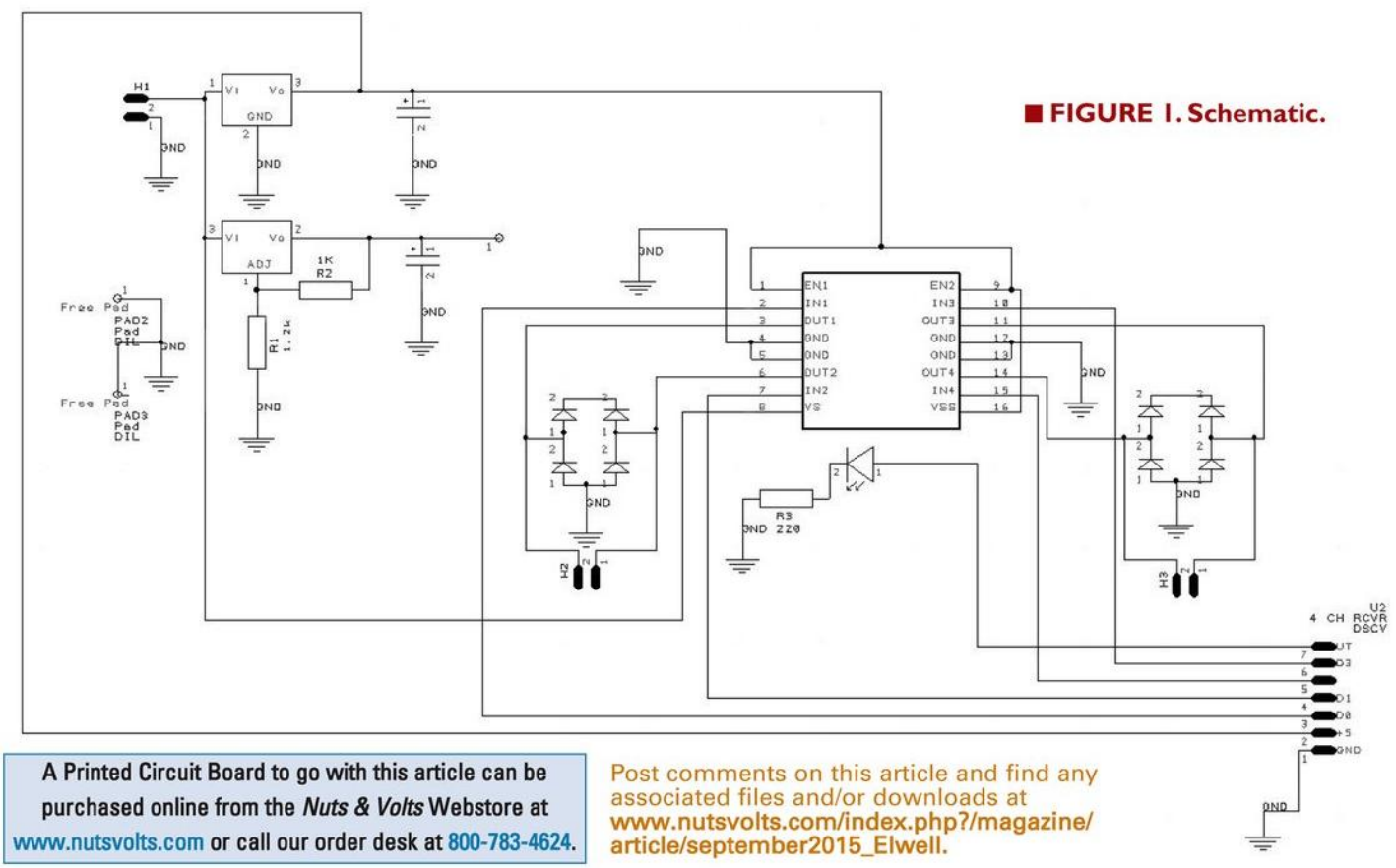
All of this is initiated by you. A four-channel remote secreted in your hand controls all the timing and functions. Those functions are:

Channel 1. Sound from module
Channel 2. Eye moves
Channel 3. Book hops forward rapidly
Channel 4. Book hops backward rapidly

Let's Get Started

There is **no** microprocessor or programming in this project. The system utilizes a double-sided PCB (printed circuit board), an L293 quad bridge, two regulator ICs, an off-the-shelf four-channel transmitter and receiver, two motors, and a sound module similar to the ones you find in a greeting card. This project is powered with a standard nine volt battery, so it is safe for everybody. Most of us electronics nerds have saved those little sound modules from greeting cards, but if you don't have one, you can just leave it out of the project or head to the dollar store.

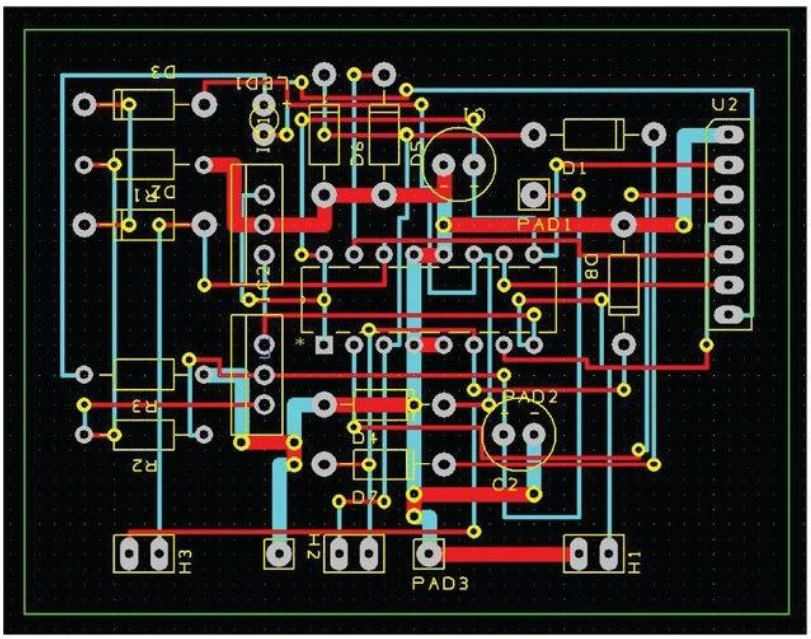
The brains are on a double-sided thru-hole PCB. **Figures 1** and **2** are the schematic and board. Keep this board as small as you can. The inside of this little book is crowded. Pay attention to the location of each part with



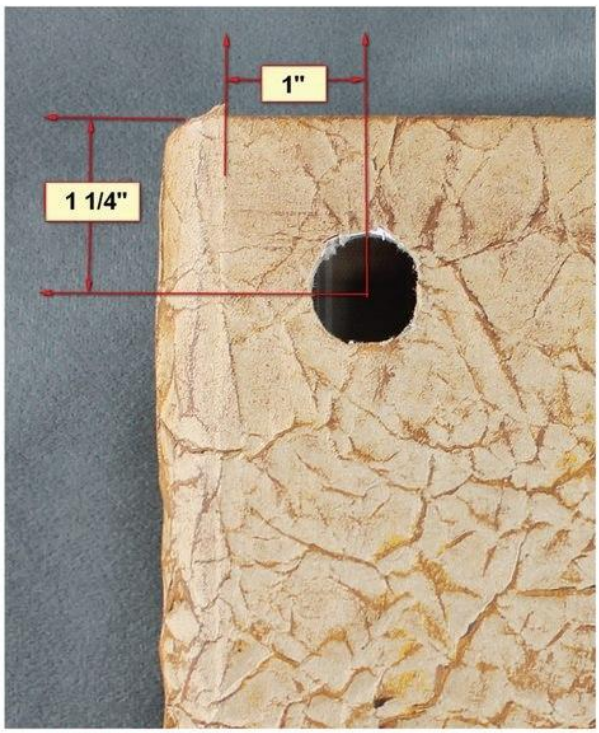
■ **FIGURE 1. Schematic.**

A Printed Circuit Board to go with this article can be purchased online from the *Nuts & Volts* Webstore at www.nutsvolts.com or call our order desk at 800-783-4624.

Post comments on this article and find any associated files and/or downloads at www.nutsvolts.com/index.php?/magazine/article/september2015_Elwell.

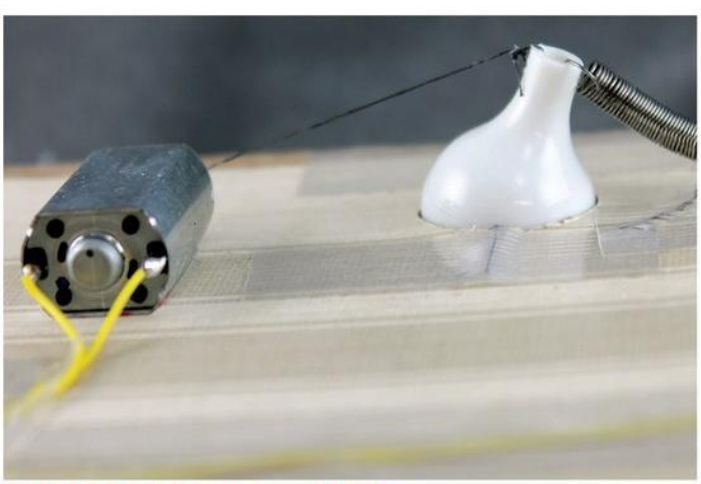


■ **FIGURE 2. Double-sided printed circuit board.**



■ **FIGURE 3. Eyeball placement dimensions.**

■ **FIGURE 4. Drilled 20 mm eyeball.**



■ **FIGURE 5. Eye motor detail.**

relation to the moving parts, and make sure they don't interfere with each other or with the closing of the book.

Construction

The PCB and schematic files were generated using the free DesignSpark software (**www.rs-online.com/designspark/electronics/**). All files are downloadable from the article link.

For purposes of this article, we will assume that the book is in the same orientation as if you were getting ready to read it. Cover opens right to left.

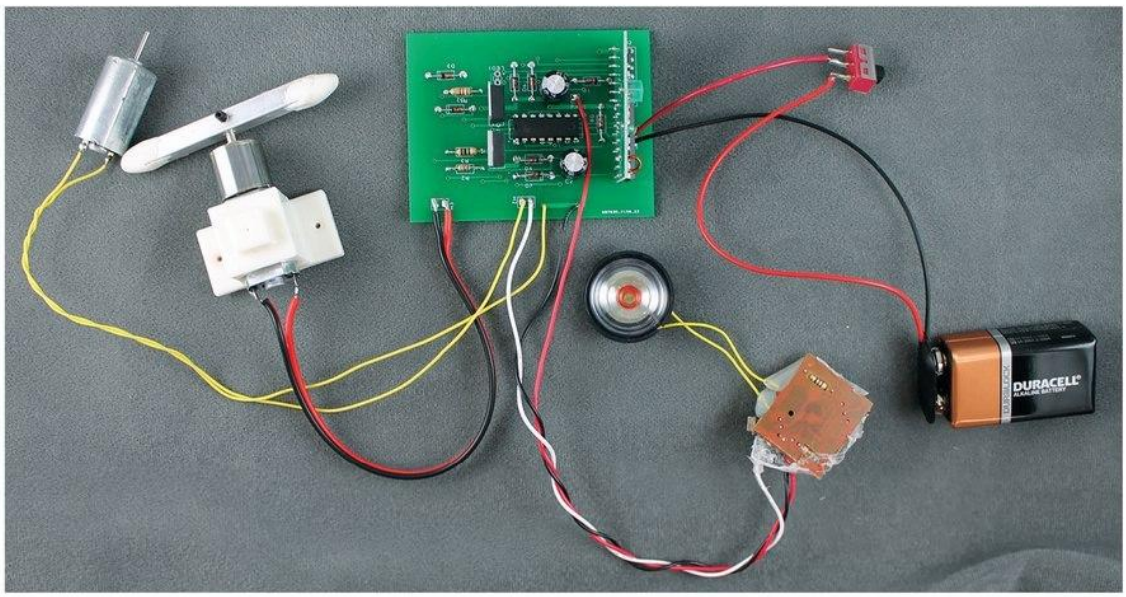
It's best to wrap the book with an "old" looking book cover first. Get some brown craft paper (grocery bag!) and cut it big enough to wrap the cover, back, and spine of the book. Then, wad the paper several times to get a good wrinkle in it. Glue it on the book avoiding the fake pages. Spray adhesive works well. (Spray the paper, not the book.) I make the paper wrap over the edges for a neat look.

Let it dry for a couple hours, then you can highlight the wrinkles with some dark brown paint on a dry brush. Go lightly or the whole thing will just be a solid color. The secret here is to wet the brush with paint, then paint a paper towel until the brush ALMOST runs out of paint. Then, LIGHTLY go over the wrinkles. When you have the look you want and the paint is dry, a coating like Mod Podge or watered-down white glue can be painted to cover the entire book for better longevity.

On the back cover (the bottom) of the book, cut a

3/4″ wide slot in the center — left to right — all the way through the cardboard, leaving only a 1/4″ of cardboard on the left and right sides. This is where the book gets its locomotion.

Drill the eye socket with a 13/16″ drill bit. I used a Forstner bit. The placement of this hole is critical so that the eye mechanism doesn't rub on the 600 RPM motor mount. After a lot of trial and error, I determined the best placement is in the upper left corner of the cover, 1-1/4″ from the top, and 1″ from the spine (where it bends). Refer to **Figure 3**.



■ **FIGURE 6. Populated and wired PCB.**

Drill the plastic eye with a 1/16″ drill bit close to the mold mark all the way through to the other side. I drill each hole separately (since the eye is hollow) to make sure they are perfectly aligned. Turn it 90 degrees and drill close to the end of the 'stem' as shown in **Figure 4**. Place the two inch piano wire through the first holes creating a hinge point. Place the eye in the hole and tape the wires on both sides with filament tape so the eye can look left to right. Take the FF-030 motor and carefully (gel) super glue it adjacent to the eye so that it does not interfere with the 600 RPM motor. Be careful that the super glue doesn't enter the motor, or you will have a paperweight and need to buy a new one. Tie a surgeon's knot with the thread (I used Fireline — strong stuff!) onto the end of the motor shaft and put a tiny spot of gel type super glue to keep it from slipping. Tie the other end of the thread to the end of the eye stem. Attach a weak spring as shown to pull the eye back after the motor has de-energized. This is a lot of commotion to get the eye to move — but it's worth it. Color the eye if you want and hook the motor up for a test. Fiddle with it, restring, fiddle — done (**Figure 5**).

Drill the center of the aluminum bar all the way through with a 3 mm drill bit. Then, turn it 90 degrees and drill a setscrew hole (#43 drill bit) for the 4-40 setscrew. Tap the setscrew hole and mount the bar on the 600 RPM motor. You now have a "propeller."

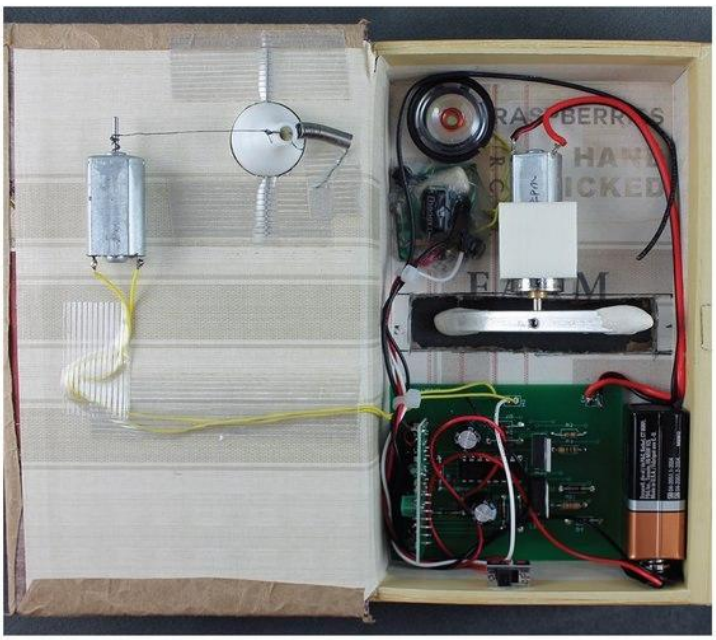
On each end of the propeller, put some Sugru or Oogoo for traction. Once dry, mount the motor so that the propeller is central in the slot. I used a 3D printer to make the plastic motor mount (the file is at the article link), but wood or cardboard will also work. Just make sure it is secure and doesn't interfere with the eye.

After populating the PCB and attaching all the external parts (**Figure 6**), place the PCB inside the box to the lower left as far as possible. Use foam tape to hold it in the box. The LM317 regulator is to output the voltage your sound module needs. Only two resistors are required to get the proper voltage. The schematic is set up for 2.9 volts. See the LM317 datasheet for the proper resistors if

PARTS LIST

Qty	Description
1	L293D dual H-bridge motor driver; Adafruit #807
1	FF-030 motor; All Electronics CAT#DCM-433
1	600 RPM 12V gear motor; Deal Extreme #323578
1	7805T220 5V regulator; Mouser 595-UA7805CKCT
1	LM317T220 regulator; Mouser 511-LM317T
2	470 µF 16V electrolytic capacitor
8	1N4148 diode or similar; Mouser 583-1N4148-T
1	100 ohm 1/4W resistor
1	1K 1/4W resistor
1	1.2K 1/4W resistor
1	9V battery clip
1	Small SPST slide switch
1	Sound module (see text)
2"	1/16" piano wire
	Fiberglass tape 1/2"
6"	Strong thread
1	Weak spring
1	Aluminum bar 1/4" square, 2-1/4" long
1	Small book box, made by Punch Studio #10808, 4-1/2" x 6-1/2" x 1-1/2"; or try Michaels, Joanne Fabrics, or book stores (could not find these online!)
1	Double-sided printed circuit board (see text for file location)
1	20 mm plastic eye
1	Four-way transmitter and receiver; **Dx.com** #148825 or Amazon 12 VDC 4 CH Rolling door remote control switch module controller set

- Brown craft paper
- Velcro™
- Foam tape
- Mod Podge or white glue
- Acrylic paint



■ **FIGURE 7. Book parts placement.**

yours differs from 2.9V.

Route the wires to keep them away from the propeller. Mount the slide switch just below the "pages" for easy access. Put some Velcro™on the side of the battery and mount it in the slot beside the PCB. Mount any sound module on the other side of the prop motor. Be careful to keep it away from moving parts. Refer to **Figure 7** for the parts placement.

Resources

Demo of book:
www.youtube.com/watch?v=JJiAmtvZmJI

Part 1 of How to Build:
www.youtube.com/watch?v=0jjNDvTGXEM

Part 2 of How to Build:
www.youtube.com/watch?v=H53bCvFIqr4

Operation

Wait for an unsuspecting victim. Remember, Channel A is for sound. When they come over to look, Channel B is for the eye movement. People are curious. They will normally reach to open the book. Timing is everything here. Use Channel C or D to make the book jump at them, then retreat. Awesome!

If you like Halloween and scaring (or startling) people, then this is for you! **NV**

By Jon McPhalen

FLAME ON, DUDE!

I know I must sound like a broken record at times, but I really do love living in Los Angeles, CA. Yes, there's traffic. Yes, there's heat and smog in the summer. Yes, the cost of living can be tough. Pushing those inconveniences aside, it's a great city filled with wildly creative people — especially in and around the entertainment industry which is where I do most of my work. Interestingly, I'm one of the few natives of Los Angeles; most people in "the business" come from across the nation and across the world. One of my friends is a guy named Matt, who's originally from Ohio and now makes his home in Los Angeles. By day he's a corporate IT expert; by night, he's a wild-eyed character creator!

Okay, he's not that wild-eyed, but some of his characters are! Matt's passions fit right into the diversity that is Los Angeles: He makes everything from crazy plush characters to full animatronic displays. From time to time, Matt is contracted to repair or upgrade props and displays, which is what leads us to the topic at hand. **Figure 1** shows Matt with his "Disco D2" prop at the San Diego, CA Comic-Con. Disco D2 is a Propeller powered prop. (You know, on second thought, Matt does look a bit wild-eyed in that picture!)

A few months ago, Matt was commissioned to add lighting to a Gort prop (**Figure 2**) owned by legendary rock guitarist, Kirk Hammett. Matt knew he would use a strip of WS2812s, and wanted to make it really interesting with a pseudo flame effect. He called on me for an assist because I'd recently coded a big project for Alliance Studio that was all about faux flames.

For the record, it is very hard to simulate fire and flames with LEDs — but here's the good news: We don't have to simulate, we only have to *suggest*. Let the last part of that sink in a moment — especially as you're planning your Halloween displays. The human mind is an amazing thing. With just a little bit of information nudged in the right direction, it will imply the rest. Our job, then, is to provide enough of the right information and the right direction, allowing the mind to handle everything else.


FIGURE 1.


FIGURE 2.

Post comments on this article and find any associated files and/or downloads at **www.nutsvolts.com/index.php?/magazine/article/september2015_McPhalen.**


FIGURE 3.

Resources

Jon "JonnyMac" McPhalen
jon@jonmcphalen.com

Parallax, Inc.
www.parallax.com
Propeller boards, chips, and programming tools

Matt Hawkins
www.likeform.com
Custom creature design and fabrication

Allow me to share a true story. While living in Dallas, I was experimenting with a single-LED flame effect. I had two candles: one (unmodified) was lit, while the second had the core removed so the wick could be replaced by a yellow/orange LED and a small SX microcontroller. I was comparing the light cast by each on a wall; I was looking for my artificial candle to closely approximate the real one.

At some point, the real candle went out. I reached for a match and instead of striking it on the box, I unconsciously touched the match head to the faux candle LED expecting it to light! After a second, I caught myself and burst into laughter. I tell this story not to poke fun at myself, but to reinforce what I said above: We only need to provide a little information and the right direction. Our minds will fill in the blanks.

One of the neatest projects that I've coded was the "League of Legends" Annie & Tibbers display built by Alliance Studio for Riot Games. **Figure 3** shows me with the display as it now sits in the lobby at Riot Games in Santa Monica, CA. Yes, it's massive. Tibbers is about eight and a half feet talk from the bottom of his feet to the top of his head. Interestingly, the whole display is controlled by a single Propeller chip.

My strategy for this project was to use the Propeller (in the form of an EFX-TEK HC-8+) as a DMX master. Small DMX "bricks" placed around the display would provide connections for hundreds of 2W and 3W LEDs. If you look closely, there are trees that appear to have embers, while Tibbers' seams are bursting with flames. With so many flame channels, I needed a simple flame generator in code — one that I could easily duplicate and modify to give different effects.

The algorithm is ridiculously simple: I keep an array of eight "heat" elements that bounce back and forth between two ranges; in my case, I use 0%-25% for the low side and 75%-100% for the high side. Bouncing is controlled by keeping a target value for each channel. Moving to a new target is a simple matter of incrementing if the current level is below, or decrementing if the current level is above. The rate of change is controlled by a third variable (speed) for each channel. For a little variety, the algorithm occasionally limits the high target value to the 25%-50% range.

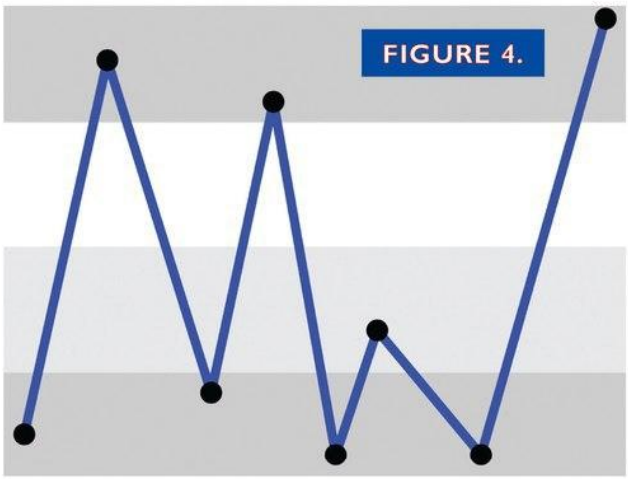
Figure 4 illustrates the behavior of the heat channel; again, there are eight of these running at the same time.

Here's the code that handles the heat generation on eight channels:

```
var

  long  stack[64]

  byte  heat[8]
  byte  tgt[8]
  byte  spd[8]


pri make_heat | mode, ch, mask

  mode := %00000000

  repeat ch from 0 to 7
    tgt[ch] := (prng.random >> 1) // 64 + 192
    spd[ch] := (prng.random & %1) + 1

  repeat
    time.pause_us(HEAT_RATE)

    repeat ch from 0 to 7
      mask := 1 << ch
      ifnot (mode & mask)
        if (heat[ch] < tgt[ch])
          heat[ch] := (heat[ch] + spd[ch]) <# 255
        else
          mode |= mask
          tgt[ch] := (prng.random >> 1) // 64
          spd[ch] := (prng.random >> 1) // 1 + 1

      else
        if (heat[ch] > tgt[ch])
          heat[ch] := (heat[ch] - spd[ch]) #> 0
        else
          mode &= !mask
          if (prng.random & %11)
            tgt[ch] := (prng.random >> 1) // 64 + 64
          else
            tgt[ch] := (prng.random >> 1) // 64 + 192

          spd[ch] := (prng.random >> 1) // 3 + 1
```

Bill of Materials

Propeller Mini	Parallax #32150
12-pin strip socket (x2)	Digi-Key 929974E-01-12-ND
WS2812 24-pixel ring	Adafruit #1586
Adjustable DC power supply	Amazon #B008BHAOQO
12V wall wart	EFX-TEK 750-00007
2.1 mm pigtail jack	All Electronics #CB-209
330 ohm resistor	All Electronics #291-330
Perfboard	All Electronics #SB-37



A quick note on style: Using a **PRI**vate method in the top object file doesn't do anything special; I use this as a marker to remind myself not to call this method in the usual way; **cognew** must be used to launch this method into its own cog.

At the top, we initialize *mode* to zeroes to force each channel into brighten mode; this will cause the heat level to increment to the target, which we initialize into the high range. The rest of this code loops through the channels, incrementing toward the target when the mode bit is 0 and decrementing to the target when the channel mode bit is 1. When a new target is set, the speed is updated as well. I tend to go with a smaller value (faster) when incrementing than decrementing, but this is purely an aesthetic choice.

Finally, when choosing a high-side target, the code does a random test and will occasionally use the 25%-50% range. Again, the technique I used for the randomized selection (lower two bits or new random value are set) is arbitrary; this is about art and aesthetics, and will take the most time to tune to your liking (art is always more time-consuming than technology).

What we have now is an array of values that are randomly bouncing between two points, and if we run these values through a PWM driver into LEDs, we can suggest flames. For the Tibbers project, I created duplicate sets of these algorithms with slightly different values. By using byte arrays, I was able to plug the heat values directly into the DMX stream.

Any of my objects that use an output buffer have a method called ***.address()*** which provides the hub address of the output buffer. With **bytemove**, we can copy the heat array directly into the DMX stream:

```
bytemove(dmx.address+0, @heat1, 8)
```

It's that easy. In the actual Tibbers code, this happens four times as there are two methods generating flame levels, and two more generating ember levels; the latter uses lower high-side values and much slower timing.

Let's get back to Matt's project and how we can put it to use in a simple Halloween prop. WS2812s require a 24-bit color value. What we need to do is convert an eight-bit heat value into a 24-bit color value. I do this with a method called ***morph()*** which uses a scaling code from the WS2812 object:

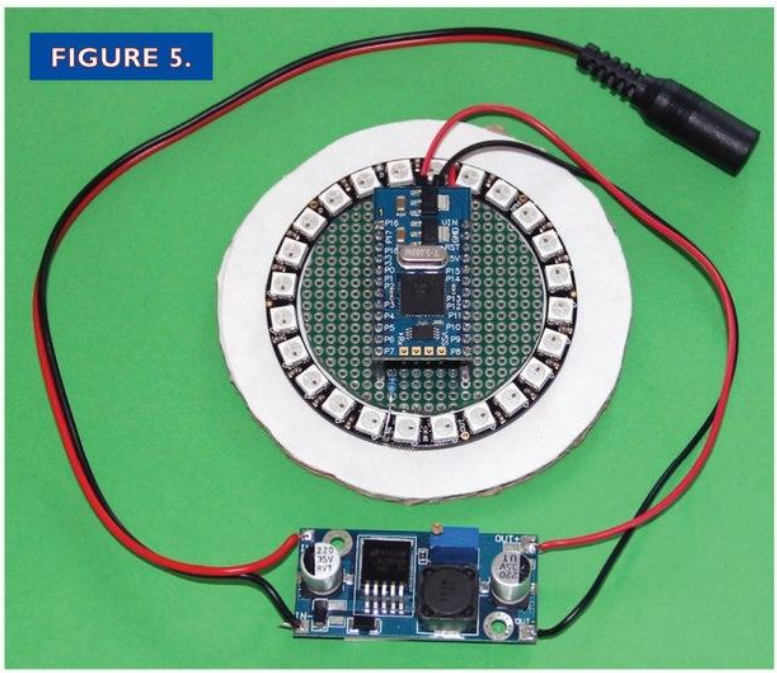
```
pub morph(c1, c2, phase) | newcolor

  if (phase =< 0)
    newcolor := c1

  elseif (phase => 255)
    newcolor := c2

  else
    c1 := strip.scale_rgb(c1, 255-phase)
    c2 := strip.scale_rgb(c2, phase)
    newcolor := c1 + c2

  return newcolor
```

This method expects two 24-bit colors and a [DMX compatible] phase value that is between 0-255. If the phase is between 1 and 254, we use it as a color mixer; below 128, the color will be biased toward *c1*; above 127, the color will be biased toward *c2*. For Matt and me, this simple two-point ramp works well. For more complex ramps, this method could be called from a **case** structure that divides the heat value into three or more ranges.

To keep the code simple for Matt's project, I also created a scrambled array of heat indexes. This allows me to have more than eight randomized WS2812s without multiple heat generators. The reason I go to this trouble gets back to the human mind: Just as it's open to suggestion, it's even better at pattern recognition.

If we have a string of 24 LEDs that is divided into three groups of eight, and each group is doing the same thing, the mind will catch the duplication and suspension of disbelief will be interrupted. We may not even be conscious of this; we simply know that something is "not right."

So, why don't I manually scramble the array? Well,



here's why: In a strip project like Matt's, I want to have each LED be its own color; in a ring project like mine, this doesn't work as well. So, what I want to do is create groups of LEDs (super pixels) to produce a stronger effect:

```
pub randomize_flames(size) | idx, used, ch,
last

  idx := 0

  repeat
    used := %00000000
    repeat while (used <> %11111111)
      ch := prng.random & %111
      if (ch <> last)
        ifnot ((1 << ch) & used)
          repeat size
            Scramble[idx++] := ch
            if (idx == N_PIXELS)
              return
          used |= 1 << ch
          last := ch
```

This code is simple, if not obvious. What we're doing is looping through the *Scramble* array, assigning a random index to the number of elements defined by size. The index can be 0 to 7 to correspond with the heat array we created earlier. Two variables are employed to ensure a proper mixing of the indexes: *used* makes sure that all indexes are, in fact, assigned before there are any repeats, and *last* ensures that the same index is not assigned to side-by-side channels.

Here's the fun part: With all of that prep, the mainline code requires just a few lines:

```
pub main | ch, color

  setup

  randomize_flames(6)

  repeat
    repeat ch from 0 to N_PIXELS-1
      color := morph(COOL, HOT, heat[Scramble[ch]])
      strip.set(ch, color)
```

This happens to be from my pumpkin project; I used six in **randomize_flames()** to convert my 24-pixel ring into four groups of LEDs ("super pixels" of the same color). Why? Well, for a ring project, having too many small colors causes them all to blend together, resulting in a "mushy" average that lacks any of the characteristics we associate with flames. By using fewer elements, we restore the sense of brightness changes and motion that suggest real candles in a Jack-o-lantern.

You've seen Matt's cool project; here's mine: I decided to upgrade my faux Jack-o-lantern from a couple years ago with a ring of WS2812 LEDs. There's nearly nothing to it: I used a Propeller Mini, a ring of WS2812s, a DC-DC power supply, and a 2.1 mm pigtail so that everything can be very modular.

If you've already used a Propeller Mini, you know that

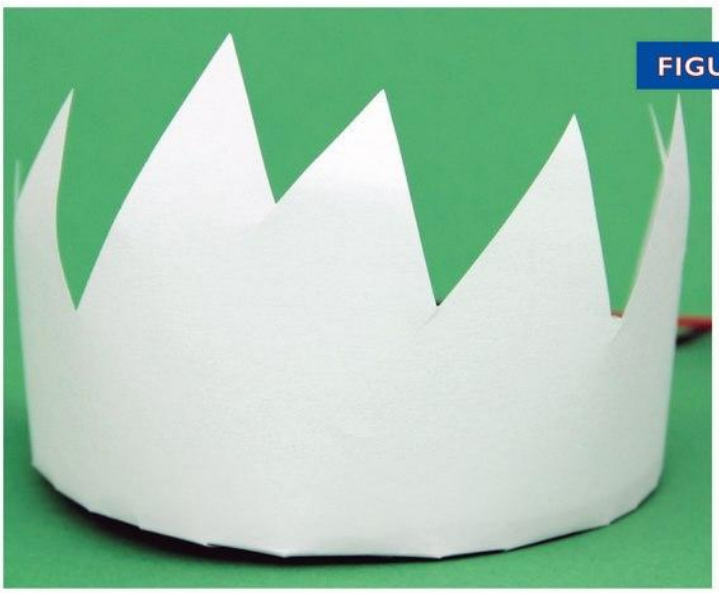


FIGURE 6.

it has its own five volt regulator. I went with an external supply so that I could safely light all 24 WS2812s, and even connect to a second ring if I desire. Yes, it's okay to connect an external 5V supply to the 5V pin on the Mini; just make sure that you leave the VIN pin floating.

I used a rotary tool to cut a piece of perfboard to fit inside the LED ring. Remember to wear a dust mask when doing this; fiberglass dust is not good for the lungs. After adding male pins (they're provided) to the Mini, I pushed the female sockets onto those pins and then centered the assembly in the perfboard. The connections are easy: 5V to the Mini and LED ring; and one pin from the Mini to the DIN pin on the ring through a 330 ohm resistor. **Figure 5** shows all the components soldered and ready for installation.

I mounted the Propeller Mini and LED ring to a piece of scrap cardboard with double-sided foam tape. To this, I attached a diffuser element made from a piece of plain printer paper. We need a diffuser because we don't want to see direct light from the LEDs; it is the variance in the indirect light that creates the effect. Don't worry, the diffuser doesn't have to be fancy (see **Figure 6** for proof!), and it only needs to cover the front half of the LEDs. Leaving the back half open allows more light to bounce around inside the Jack-o-lantern.

This was really simple. In a couple hours, I upgraded my Jack-o-lantern from a few LED wicks to a color-controlled flame effect. My Jack is pretty standard, so the colors (dark red to bright orange) I chose for the flame levels are standard as well. Remember, though, this is all about the art. If you have a cool purple pumpkin, you can give it some cool purple flames instead.

Finally, your display doesn't have to be as docile as mine. You could connect a PIR or mat switch to respond when a trick-or-treater comes near. The construction phase will take a couple hours. It's the playing with the code phase where you'll have the real fun and spend the most time. You've got a bit of time before the big night, so get to it!

Happy Halloween, and here's to spinning up a lot of cool creations with the Propeller chip! **NV**

Single Chip Audio Spectrum Analyzer Makes Singing Easier

Autumn is my favorite time of year. As the days grow shorter, the blistering heat of summer in Southern California is coming to an end, and it's a great time to be outside working on my very popular Scary Lane yard haunt for Halloween. This is a perfect hobby for my creative side, and I get to have a little fun with the neighbors. Each year, the haunt improves with new scenes and props to scare and delight the trick or treaters. About five years ago, I wanted to include a new audio animatronic of a trio of singing skulls that would fit nicely into the old west ghost town theme.

FIGURE 1. Ready-to-go skull with a nice aged look.

The core of this project involves three talking/singing skulls that I picked up for about $10 each from Walmart's seasonal Halloween section (see **Figure 1**). A simple lobotomy of the skull's built-in controller put an end to the corny, "I ain't Got Nobody" song and gave access to the jaw movement system. In place of a costly servo, these skulls use a simple motor to open the jaw, with a spring to close it again when the power is turned off.

The design consists of a power supply, band detection circuit, MP3 player, and CPU/driver board. A CHI035A PICAXE-18M2 power project board was used to control the MP3 player, the three jaws, and to show lights. The right channel of the MP3 player was fed into the three-spectrum band detector and the left channel goes to both left and right audio outputs (see **Figure 2**).

By Steven R. Bjork

Post comments on this article and find any associated files and/or downloads at
www.nutsvolts.com/index.php?/magazine/article/september2015_Bjork.

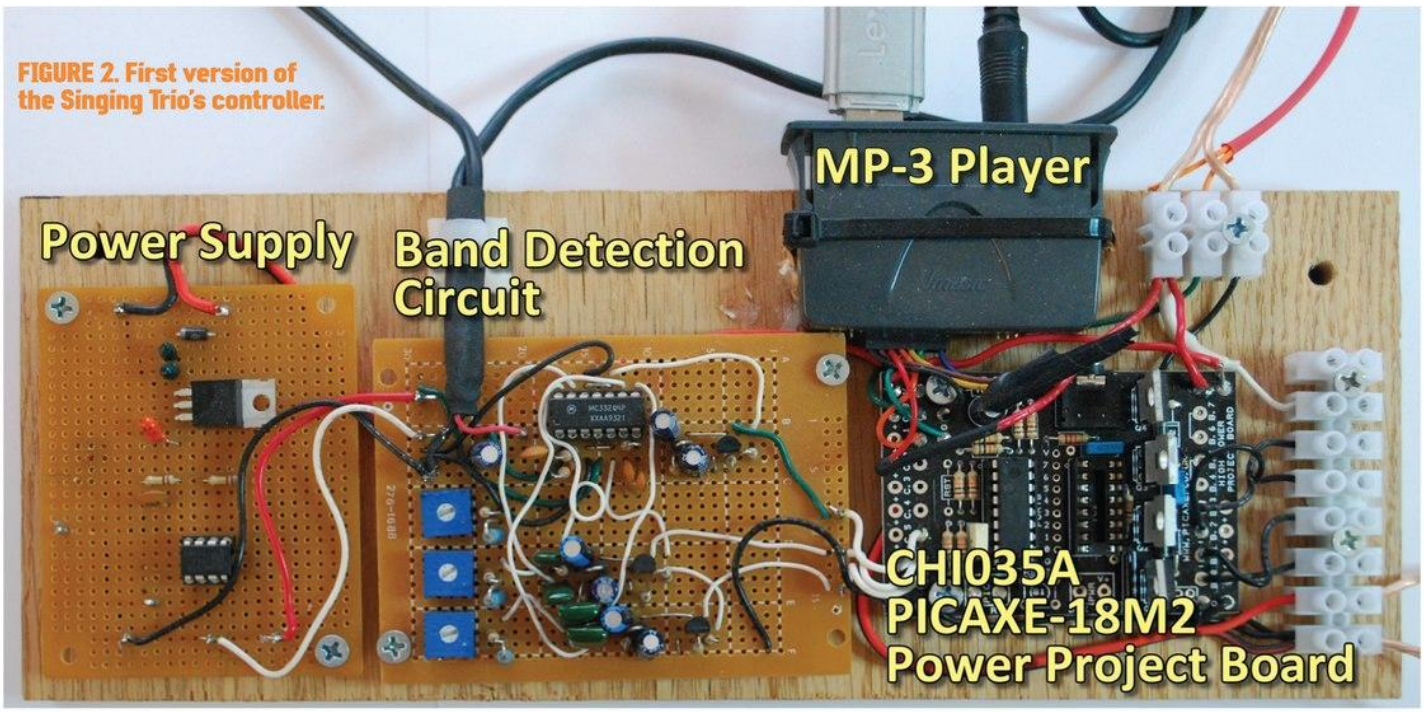


FIGURE 2. First version of the Singing Trio's controller.

Using a two-track recording for audio playback and multi-tone control would synchronize the skull's jaw to the audio. The Audacity audio editor was used to build the sound track, along with the three jaw tone tracks. With a little trial and error, the three skulls were moving their jaws in perfect sync with the song.

The project took about a week to build and has worked (nearly) perfectly season after season (**Figure 3**).

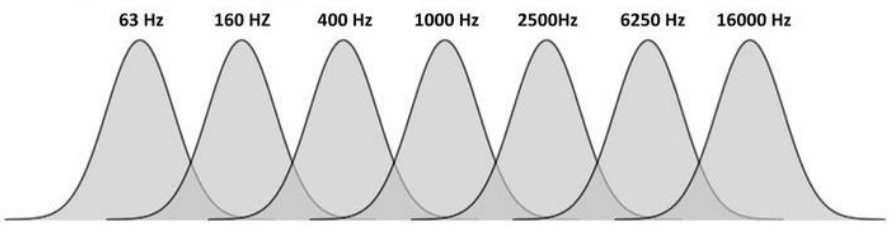
FIGURE 3. The Singing Trio mounted in the haunt.

Doing More with Less

The "Singing Trio" was a good and simple design, but what if I wanted to build another prop using audio to control it? I was not looking forward to wiring up another 50 plus parts for a new spectrum band detection circuit. There must be a better way to analyze the audio in my future projects. Besides, I can never leave well enough alone.

The very next year, Mixed Signal Integration introduced their "7-Band Graphic Equalizer" (MSGEQ7). Don't let the name fool you, however, since it's really a seven-band audio spectrum analyzer. (The chip is used to run graphic equalizer displays in consumer audio gear and the reason MSI picked that name.) With seven bands, this audio spectrum analyzer has a wide variety of uses, from tone detection to real time audio analyzation for controlling servos and other devices. **Figure 4** shows the

FIGURE 4. Frequency response.



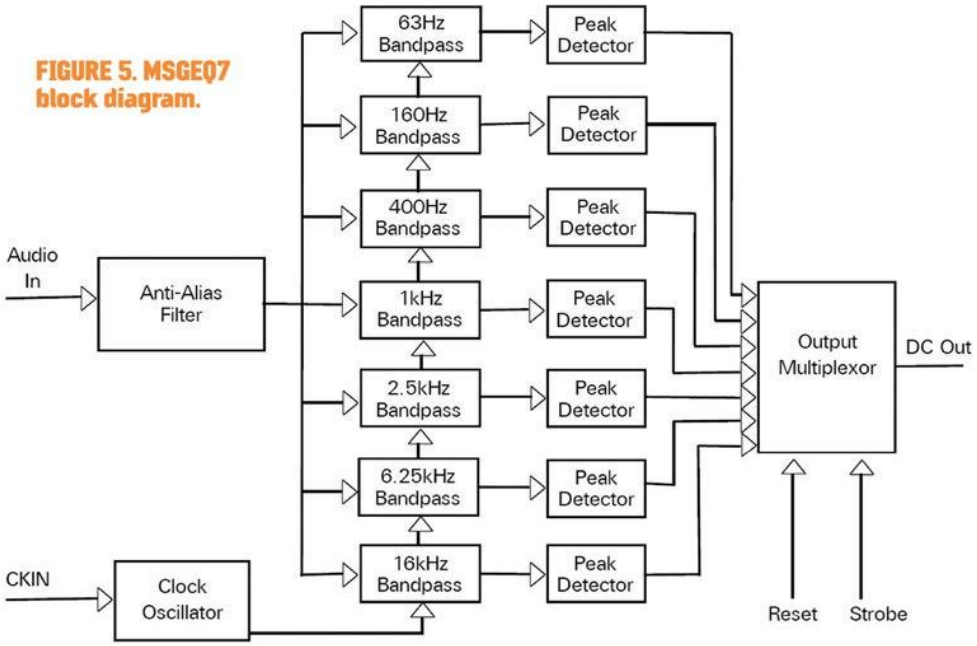


FIGURE 5. MSGEQ7 block diagram.

Haunt Hackers was created by Steve Bjork and Steve Koci to help their fellow haunt enthusiasts with Do-it-Yourself (DIY) projects and prop controllers. With the right skills and a willingness to build, test, and debug your projects, you too can bring life to your haunt on the tightest of budgets.

To keep the cost low, Haunt Hackers does not charge for the designs or software. For easier building, many of the projects have a printed circuit board available.

It's here that you can read more about the super controller, Banshee and how it uses two MSGEQ7s for monitoring the playback audio to great effect.

peak points of the seven spectrum bands.

The block diagram in **Figure 5** shows the overall functions of the chip. The audio input is sent to the anti-alias filter and then to the seven bandpass filters with peak detectors. An output multiplexer is used to select the spectrum band to send out of the chip. This output level ranges from zero to just under the power supplied to it.

This eight-pin chip only needs the help of two resistors and four capacitors to get the job done (**Figure 6**). It can run safely from 2.7 to 5.5 volts, so it works with both 3.3 and five volt logic systems. However, five volts provides the best performance.

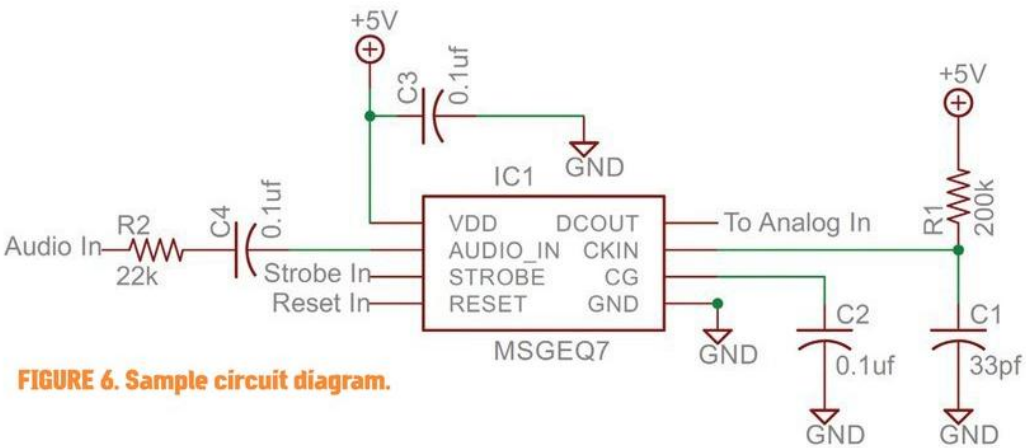
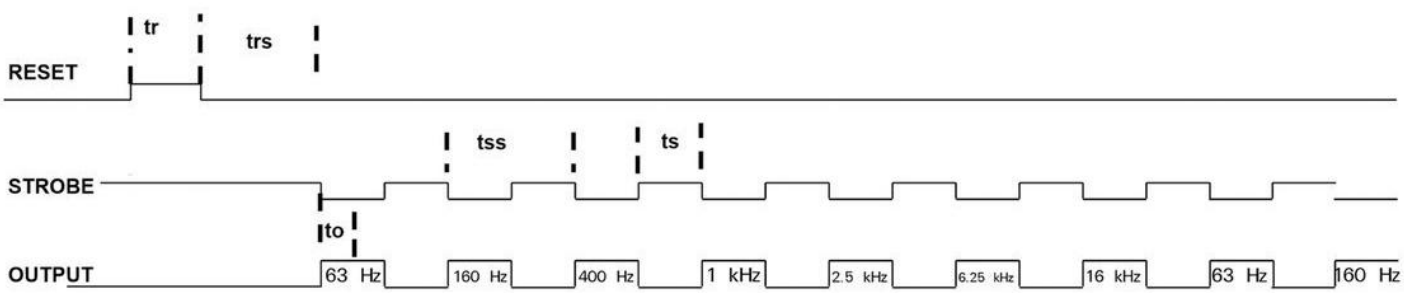


FIGURE 6. Sample circuit diagram.



tr - Reset Pulse Width		200 nS	min
trs - Reset to Strobe	Delay	72 uS	min
ts - Strobe Pulse Width		18 uS	min
tss - Strobe to Strobe	Delay	72 uS	min
to - Output Settling Time		36 uS	min

(with Cload = 22 pF and Rload = 1 Mohm)

FIGURE 7. Strobe timing diagram.

Scary Lane Haunt (ScaryLane.com) started out as a simple home Halloween display over a decade ago. It wasn't long before it was the must-see attraction for the Whittier community in Southern California.

About six years ago, we added a two-story Ghost Town façade that hides the house and has plenty of places for audio-animatronics, skeletons, props, and set dressing. The key to making this haunt great has been the use of electronics to run the show.

Using more than a dozen custom-designed show and prop controllers, the Ghost Town literally comes alive on its own to entertain the neighborhood. The controllers time the fog machines, the lighting, and other effects based on the time and the weather conditions. Even the audio-animatronics changes based on the time of day or if it's the big night of Halloween.

This year, I will be adding more sensors along the side walk so the show elements know where people are. Just imagine ... a person is walking by and the talking skeleton follows them. I will post the videos of the freaked out passers-by.

Listing 1.

```
;The following lines of code are used to initialize the I/O lines
;going to the MSGEQ7 chip.  Use at the start of the program...
       High MSGEQ7Strobe    ;initialization output to the MSGEQ7
       Low  MSGEQ7Reset

;This subroutine is used to reads the 400HZ, 1000 HZ and 6.25 KHZ bands
;and opens or closes the jaw based on that band's output level.

ReadEQU:
       pulsout MSGEQ7Reset,1;strobe the reset pin for 10ms
       pauseus 8            ;wait 80us for reset to end

       gosub SkipBand       ;skip 63 hz band
       gosub SkipBand       ;skip 160 hz band

       gosub GetBand        ;get the 400 hz band
       if Jaw_Temp>threshold then  ;higher than threshold voltage?
              High Jaw1motor;yes, then open jaw
       else
              low Jaw1motor ;no, close Jaw
       endif

       gosub GetBand        ;get the 1000 hz band
       if Jaw_Temp>threshold then  ;higher than threshold voltage?
              High Jaw2motor;yes, then open jaw
       else
              low Jaw2motor ;no, close Jaw
       endif

       gosub SkipBand       ;skip 2,500 hz band
       gosub GetBand        ;get the 6,250 hz band
       if Jaw_Temp>threshold then  ;higher than threshold voltage?
              High Jaw3motor;yes, then open jaw
       else
              low Jaw3motor ;no, close Jaw
       endif
return

GetBand:
       low MSGEQ7Strobe     ;Strobe line Low
       pauseus 4            ;wait 40.0 us till data is good
       readadc MSGEQ7Data,Jaw_Temp ;get the band's audio level
       High MSGEQ7Strobe    ;Make Strobe line high
       pauseus 4            ;wait 40 us
       return               ;now safe to strobe again.

SkipBand:
       pulsout MSGEQ7Strobe,4 ;strobe line 40.0 us
       pauseus 4            ;wait another 40.0 us
       return               ;now safe to strobe again.
```

The Interface

The MSGEQ7 uses the reset and strobe inputs to control the multiplexer for spectrum band selection. Each time the strobe line is pulsed, the output will select the next band with the reset line starting a sequence back to the first spectrum band. This interface only needs two output lines and one analog input line, and is a piece of cake for just about any controller (like the Raspberry Pi, Arduino, PICAXE, or BASIC Stamp) to handle.

Timing is Everything

The strobe timing diagram (**Figure 7**) shows how to use the reset and strobe lines to select the 63 Hz spectrum band and move along to the other six. To start reading the first spectrum band, the reset and strobe lines need to be set high. About 200 or more nanoseconds later, the strobe line should go low. Wait another 72 or more microseconds before changing the strobe line to read the analog input for the seven bands.

When reading the bands, the chip responds to the high to low transition on the strobe line, but there is a short delay of 36 microseconds before the output is stable. The microcontroller can read the current level via the controller's analog input line.

Now, return the strobe line back to its high state and wait another 36 microseconds. Continue using the strobe line till all the bands have been read.

Almost Time

Trying to get timing down to the microsecond can be

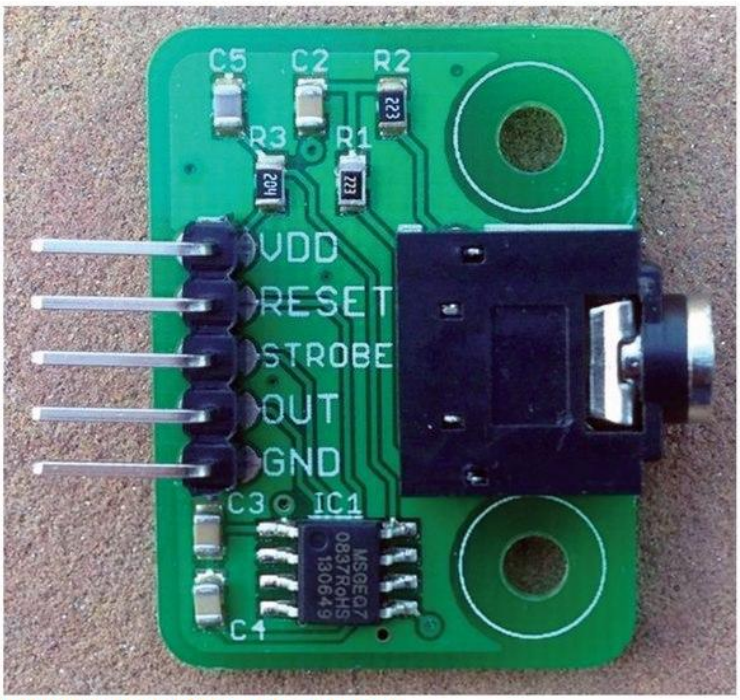


FIGURE 8. MSGEQ7N breakout board.

tough for some microcontrollers. In the case of PICAXE Basic, the *pauseus* command only pauses in 10s of microseconds. So, a *pauseus 2* is really a 20 microsecond pause. At one point, I needed a pause for 72 microseconds and the closest pause (without going under) was *pauseus 8* for 80 microseconds. All the timing in **Figure 4** is the minimum; larger delays can be used. The sample code for reading three of the seven bands is shown in **Listing 1**.

The *ReadEQU* subroutine selects the jaw's band by skipping or getting the bands in order, and opens the jaw if it detects a tone. If the subroutine is called 30 or more times a second, then the skulls will stay in sync with the music.

Resources

Singing Trio video
www.HauntHackers.com/video

PICAXE IDE and manuals
www.picaxe.com

Thor 2 and Banshee controllers
www.HauntHacker.com

MSGEQ7 chips
eBay.com and **SparkFun.com**

MSGEQ7N breakout board
eBay.com

In the years since I built the original Trio of Singing Skulls, I've created a line of controllers for the haunt industry. My Thor 2 Thunder and Lighting control has all the parts (but one) needed to run the new singing skulls. Of course, the missing part is the MSGEQ7 chip.

While the MSGEQ7P ("PDIP" version) runs from $2 to $5 depending on the source, I opted for the "N" surface-mount version on a $12 breakout board from a vender on eBay (see **Figure 8**). This made it easy to mount and hook up the MSGEQ7 to the Thor 2 board (**Figure 9**).

In no time, the new version of the Trio of Singing Skulls was up and running. I really like how clean and compact this new version turned out. Best of all, there is no more drift in the analog tone detection circuit from hot

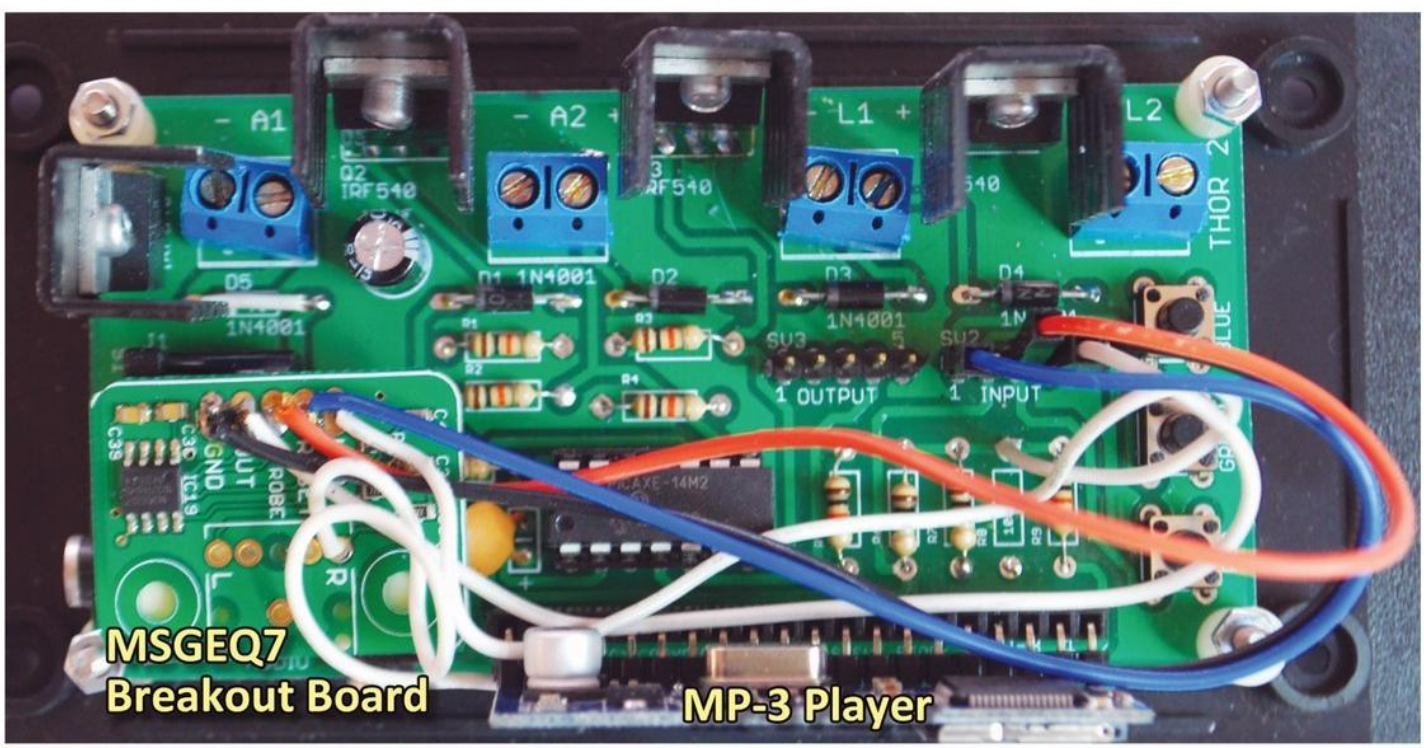


FIGURE 9. New controller with MSGEQ7N breakout board.

or cold weather. There is a video of the Trio singing over on the Haunt Hackers website at **haunthackers.com/video**.

More Projects Using the MSGEQ7

Since the MSGEQ7 was easy to use, I added two of them to my next controller project – Banshee! Why two MSGEQ7s? Well, I needed one for each audio channel. This way, I can monitor the levels of the left and right audio for a dual talking skull system without the need of a prerecorded tone track. At only $2 each, they are not only cheap but a powerful addition to this new haunt controller.

You can read more about building a Banshee for yourself at the Haunt Hackers website. **NV**

With more than 40 years of experience as a computer engineer, Steve Bjork has developed over 500 products from PCs, video games, and the Web. His skill set includes carpentry, set design, and lighting for movie studios and live productions. His entertainment background ranges from stage magician to Technical Director on live productions. Over the past decade, Steve has turned his passion for Halloween into a line of prop controllers and haunt devices. Learn more about Steve and his projects at ***HauntHackers.com****.*

Haunting 201 – Thirteen Electronic Projects to Elevate Your "Scare Game"

Continuing with the idea from last year's Halloween issue, we've compiled a list of 13 ghoulish ideas that will help take your haunt to the next level. However, this year's list is all about electronics projects!

My purpose here is not to give detailed instructions for a particular build, but rather to showcase a variety of different projects. I'll include links to pictures, videos, web pages, or forum threads that will give you more information if you're interested in pursuing a particular project further.

We'll use the same rating system as last year to help you decide on which projects you'd like to tackle. Even the most basic projects may require at least a working knowledge of computers, basic soldering, or electronics skills, and the ratings will take that into consideration.

Hopefully, you'll find at least a few new ideas on the list which you can incorporate into your displays.

1. Ghost Steps – $$$ and

This effect creates an eerie series of glowing footprints that can be adapted to fit your needs. The current basic version is free, with plans for a more advanced but paid version in the works. These footsteps can be projected on to any surface and can be adjusted to move through your display. You are able to add more feet, rotate, and move them, as well as start and stop them wherever you please. The footsteps fade in and out, creating the illusion of a ghostly presence moving through your haunt. It is a fantastic addition – especially to a graveyard scene. This effect does require a projector, which is why it earns three dollar signs. However, if you already own one, then it can be free (**Figure 1**).

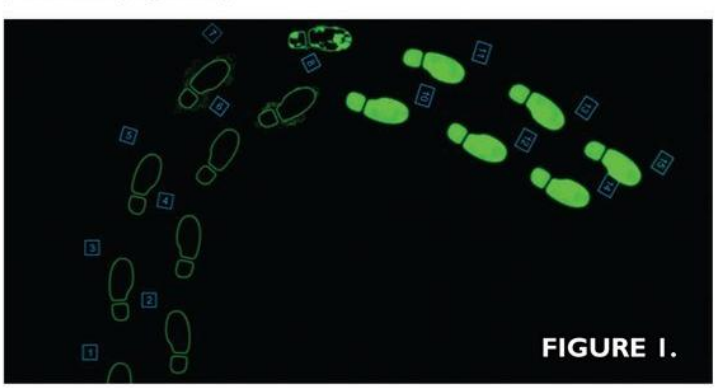


FIGURE 1.

By Steve Koci

Post comments on this article and find any associated files and/or downloads at **www.nutsvolts.com/index.php?/magazine/article/september2015_Koci.**

$ - Under $50
$$ - $50 to $100
$$$ - $100 to $200
$$$$ - Over $200

The "spidey rating" gives an indication of how much work it would take to put a project together. One spider is given for something easily assembled, where four spiders indicate a more involved build.

2. RC Car Prop — $$$ and 🕷🕷

How scary would it be if you could control a prop such as a spitting spider to chase after your targeted trick or treaters? Well, it's possible with the use of a remote controlled car chassis and wireless controller. You begin by either removing the car body from an existing frame or just purchasing a base robot frame. This now becomes your platform for attaching your terrifying spider or scary ghoul. You can let your imagination soar and add whatever scary creature you desire. I particularly like the fact that I have the freedom to choose exactly when to deploy it, and can select who to scare next. This prop can be a bit terrifying, so I prefer to avoid exposing the youngsters to it. We want them to come visit us again next year! You can see how I put my version together in the Do-It-Yourself Animatronics column in this month's issue of *SERVO Magazine* (**www.servomagazine.com**; *Nuts'* sister publication). Check mine out in **Figure 2**.


FIGURE 2.


FIGURE 3A.

3. Kit74 — $$ and 🕷🕷🕷

Do you have an old laptop sitting on a shelf gathering dust, just looking for an opportunity to be useful again? Well, you can repurpose it as a haunt controller by hooking it up to this oldie but goodie relay board. This setup is still used to run up to eight relays for many projects, including running lights up a singing pumpkin display. I found that Vixen — the free light animation software often used for holiday lighting displays — was perfect for this task. It allowed me to program my song selection and sync the music to light up my pumpkins. I know we are often enamored by new and exciting products, but there's no reason to avoid some of the old standbys. This system can allow a tremendous amount of control for a variety of props. I still use mine and the singing pumpkins are always a hit (**Figures 3A** and **3B**).

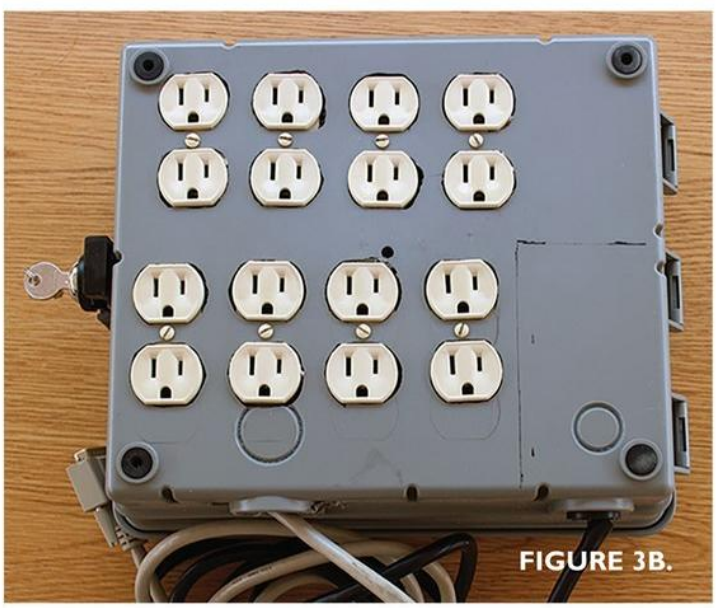


FIGURE 3B.

4. Haunt Lighting — $$$$ and 🕷🕷🕷🕷

As haunters, we spend considerable time and money in preparation of our spooky Halloween displays, often without giving adequate thought to how we are going to light them. Fortunately for us, our friends who put on the synchronized lighting displays for Christmas have solved many of the issues that we often face. They have created many do-it-yourself controllers that allow you to choose the protocol to fit your needs. I use DMX and have built a series of controllers that I use in both my Halloween and Christmas displays. I have found that the DIY Renard controllers suit my needs, and have been very reliable units with lots of help available from Christmas websites. I use Vixen software for this too because it works equally as well for this as it does for my singing pumpkins. As the technology becomes more mainstream, other software choices are now being offered. Search around for one that offers the features you need and fits within your budget (**Figures 4A** and **4B**).

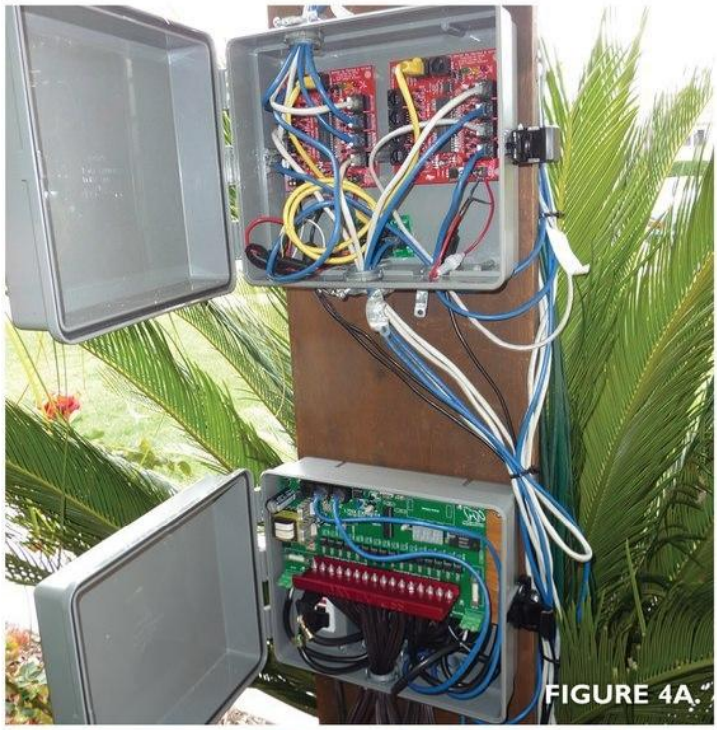


FIGURE 4A.

FIGURE 4B.

5. Raspberry Pi — $$$ and

One of the newer kids on the block as a haunt controller is the Raspberry Pi. It has created a lot of interest for those of us looking for another device to add to our toolbox. The chance to have a fully functioning computer available is very exciting!

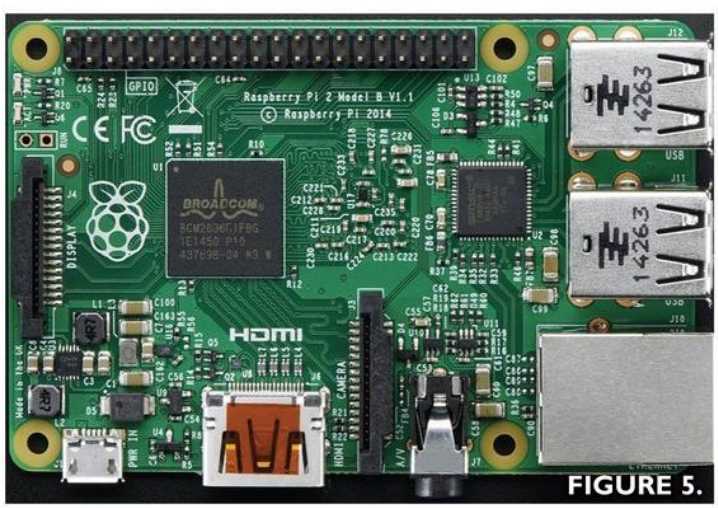


FIGURE 5.

One of the features that has me the most excited is its ability to trigger and play high definition video. The increasing addition of more and more premade videos has created a need for a reasonably priced, programmable, and triggerable video player. There are several fine commercial models that fit our needs, but they can be expensive — especially if you're running several projections. I've included in the **Resources** a couple of tutorials to set up your Raspberry Pi, as well as some sites to check out for a variety of commercial Halloween projections (**Figure 5**).

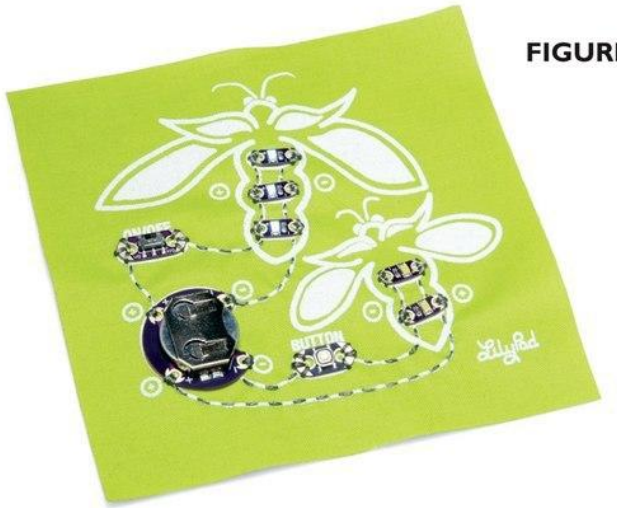


FIGURE 6.

6. Wearable Illuminated Clothing — $ and

Is your Halloween costume becoming worn and dated? Are you looking for something to spruce up and set your costume apart from all the others? If so, then you should look at the wearable lighting kits available from SparkFun. I was first introduced to this technique at a seminar at the HauntX Halloween convention. It was taught by Dia Campbell (the textile specialist at SparkFun) who was able to lead even me — a total sewing novice — to achieve a working kit. These are easy to assemble and program, and take only basic sewing skills to complete.

The effect, however, can be phenomenal! They stock an extensive assortment of components and kits available to suit your requirements and budget. Check them out and add this truly unique detail to your costume this year (**Figure 6**).

7. Triggered Fog — $ and

There's nothing that adds that spooky feeling to a

display as an ample amount of well placed fog. One of the biggest drawbacks of most commercially available machines is the ability to deliver the fog at just the right time. Most machines only have a limited amount of fog available before they need to recharge, so you want to make the most of it. A fog remote that can be fired only when your trick or treaters are in exactly the right spot is the ideal solution.

With a few simple modifications and additions to your existing remote, you can incorporate a passive infrared sensor to accomplish this. The ability to better manage your fog machines may allow you to get by with a fewer number, saving money and power – both of which can be in short supply on the big night. I think every haunt would benefit from this type of control to maximize the disbursement and timing of this valuable effect (**Figure 7**).

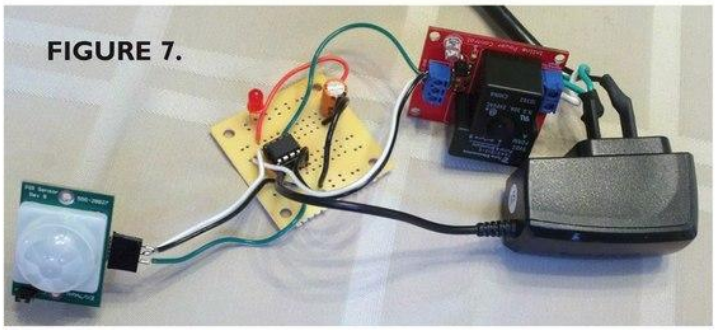


FIGURE 7.

8. Vacuum Forming – $$ and

How many times have you been working on a project and wished you could reproduce a part? Oftentimes, we only have a single item but need multiple copies to complete a build, or our original is too heavy and we require something lighter. Building and operating your own vacuum former can solve several of these problems. There are many examples of DIY models available, and they can be built with ordinary tools we already have. A variety of plastics can be used to mold a piece to fit your needs.

With 3D printers being all the rage these days, we may overlook the usefulness of the vacuum former. Once you complete the construction of your machine, creating copies is fast and inexpensive, and no computer or programming skills are required. One day, I'll own a 3D printer, but for now, the vacuum former fits the bill just fine (**Figure 8**).

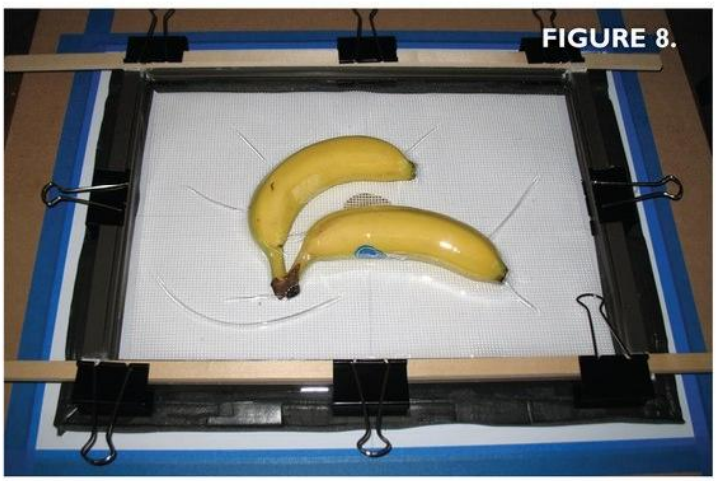


FIGURE 8.

9. Banshee PICAXE Controller – $$ and

I had the pleasure of working with fellow Halloween enthusiast, Steve Bjork on a previous project who is the real brains behind this new board design. The Banshee board will take many of the DIY printed circuit boards we now use and allow the prop builder to replace them with a single board. This board uses the 40X2 PICAXE chip as its brains, and includes eight servo outputs, four MOSFET drivers, a real time clock, and the ability to add a stereo audio player.

This board will – among other things – allow for the user to have two three-axis skulls talk to each other. (See Steve's article in this issue.) Like the previously designed

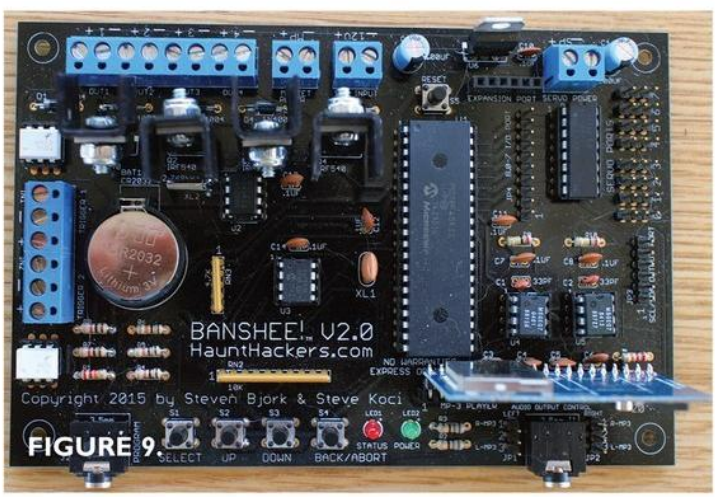


FIGURE 9.

Frankenstein board, the Banshee will walk you through the setup of the board with audio prompts stored on an SD card and played by the audio board.

The last several years have seen a rapid improvement in the number and functionality of DIY controllers, allowing the home haunter to program their characters and scenes just like the professionals. This board will certainly enable you to take your haunt up a notch (**Figure 9**).

10. Magic Mirror – $$ and

How would you like to be able to interact directly

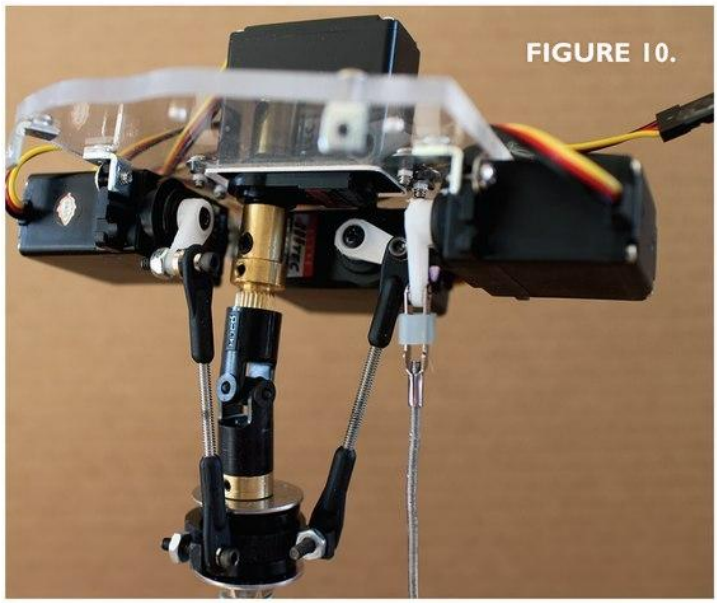


FIGURE 10.

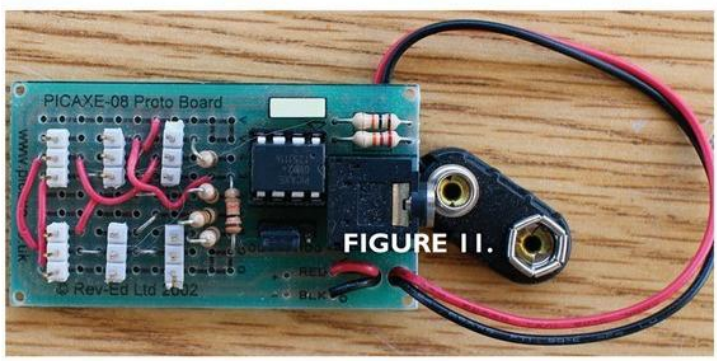


FIGURE 11.

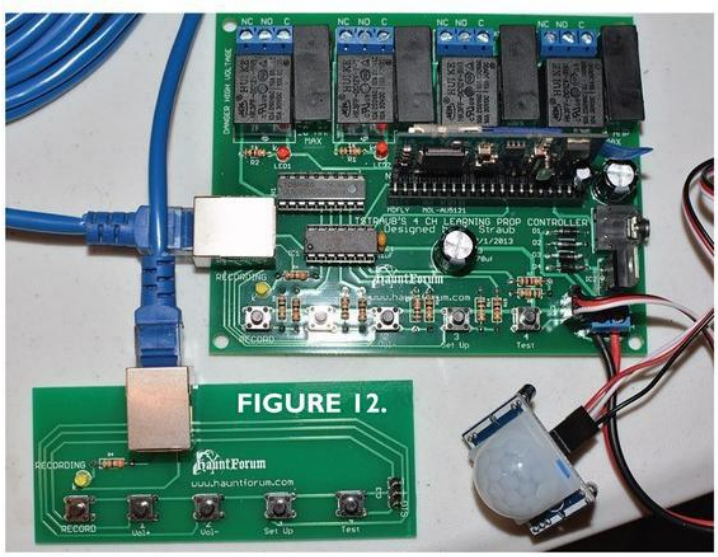


FIGURE 12.

with your guests? The Magic Mirror will provide you with the opportunity to do just that! You use a microphone and computer keyboard to add the audio and to control the actions of your choice of characters. The products are very affordable — only $15 each. They even offer a free version, although it uses text files to manipulate its behavior and is a little more complicated to control. This effect works best if you build it into a facade, and the site offers suggestions on how to create the perfect design.

A little practice learning to puppeteer and you'll soon be amazing your visitors as you converse with them. This feature can certainly become the centerpiece of your display (**Figure 10**).

11. Very Low Cost PICAXE Controller — $ and

This was where my interest and early education in the world of Halloween electronics began. Mike Langensiepen put together an outstanding beginner's project using the 08 PICAXE chip which allows you to get started building and programming your own controllers. His instructions are easy to follow, and the project cost will satisfy even the most budget conscious builder. It will allow you to get your feet wet and complete a project to build your confidence.

My success with this build encouraged me to continue to learn and take on more complex designs. Don't be fooled into thinking that this is just a practice project, however. Once completed, this board can be integrated into many different props, allowing you to control them to fit your haunt needs (**Figure 11**).

12. Four-Channel Button Banger Board — $$ and

This board was designed by Tyler Straub, but is commonly referred to as the Scuba Board after Tim Thompson who helps distribute them. These boards are available as either bare, solder-it-yourself printed circuit boards or Tim will populate the boards for you at a very reasonable cost. Prior to this board being available, I designed my own two-channel button banger, but now reach for this board for the majority of my applications which require this style of controller. Tyler has incorporated a variety of useful features which I truly appreciate. These include reverse voltage protection on the input and a separate wired remote that allows you to program the board from a safe distance from a fast moving prop.

One huge advantage to using a board such as this is that there is no programming required by the end user. You set up your sequence by simply pushing the buttons at the desired times. By following the voice prompts, you select your trigger type and delays again by just pushing the buttons. This type of controller is highly recommended for someone getting started using prop controllers or without the programming knowledge needed for some of the more complex controllers (**Figure 12**).

13. Projection Mapping — $$$$ and

The availability of useful projectors for display purposes has created a wide variety of opportunities for the home haunter. One of the most exciting is using your entire house as a canvas. You then use this backdrop to project a multitude of images to fit whatever design you are able to dream up. This effect requires some computer knowledge and the use of a short throw projector, but is very achievable especially now that the secrets to accomplish it are being revealed. This haunt concept can be especially attractive for someone who is unable or unwilling to set up and tear down a traditional display. Once you have your program put together, it is a simple process to get it running. You pull out a table which is placed in the necessary location, run some power, and plug your computer and projector in. Hit play and your house turns into a fantastic show! At the end of the night, you simply turn things off and have everything put away in minutes. That sounds like a great idea to me (**Figure 13**)! Happy Haunting! **NV**

FIGURE 13.

Resources

1. Ghost steps forum thread at **http://tinyurl.com/q37kavw** and to the website **http://tinyurl.com/ok6a7uw**

2. RC car prop video at **http://tinyurl.com/nn8mvou**

3. Kit74 forum thread at **http://tinyurl.com/pber8gs** and where to purchase at **http://store.qkits.com/moreinfo.cfm/QK74**. Also, a video at **http://tinyurl.com/onpxvh9** and Vixen software at **www.vixenlights.com/**.

4. A forum thread tutorial at **http://tinyurl.com/ol2ybt9** and a Halloween light show video at **http://tinyurl.com/pdffzzp**, plus Renard controllers at **http://tinyurl.com/pfcxse3**.

5. A forum thread on setting up the Raspberry Pi for playing video at http://tinyurl.com/op9xqhs and **http://tinyurl.com/pcyew3a** with some to try at **http://tinyurl.com/on64qrj**.

6. An intro tutorial on using E-Textiles at **http://tinyurl.com/numw5ju**

7. Building a motion sensing fog trigger at **http://tinyurl.com/o7q6mtf**

8. How to build a DIY vacuform at **http://tinyurl.com/pxlxyc6**

9. Get a Banshee of your own at **http://tinyurl.com/oy3tz5h**

10. The Magic Mirror can be found at **http://tinyurl.com/7aojcx2**

11. How to make a VLC at **http://tinyurl.com/o86nkpn**

12. Details of the four-channel button banger build at **http://tinyurl.com/qbutm83**

13. A projection mapping tutorial at http://tinyurl.com/oa7tca5 and video at **http://tinyurl.com/nk2w8gs**.

Adding Color to the Commodore PET

The year was 1977. I was in my early teens, and I happened to be at the mall. Right there in the department store I saw it ... the Commodore PET 2001. It sat on a counter all by itself with its small built-in screen, keyboard, and tape drive, cursor blinking, just begging to be played with. I pecked at the keys and they magically typed numbers, letters, and strange symbols. I remember it like it was yesterday. I was hooked. That was the exact moment I knew I wanted to be a computer geek. A few years later in high school, our computer room had six PET 4032s and one lone 4040 disk drive. I spent many hours programming and playing games on those machines, and I've been a Commodore fan ever since.

The Commodore PET 2001 was one of the very first plug-and-play home computers on the market. With its built-in 40-column monochrome screen, cassette deck, keyboard, 6502 processor, and 4K of RAM, it was ready to use right out of the box. Over the years, the PET was incrementally updated, but through its life remained fundamentally the same. It evolved from a hobby machine into a business machine, trading its 40-column screen and tape drive for a more business-friendly 80 columns with dual disk drives and even hard drives.

Commodore's CEO, Jack Tramiel knew that the future was with the low-end home market, so Commodore started thinking about making a color computer of some kind. They started with the PET as a base and built prototypes along the way using various technologies, but eventually decided to release completely new cost-reduced machines such as the VIC-20 and C64, using custom designed chips from their MOS semiconductor operation.

Fast forward to today. I've become a retro computer collector, re-learning everything about those old machines. I've been a computer programmer and IT guy over the years, and I've tinkered with hardware and electronics but not seriously. I decided I wanted to get into it more as a hobby, and wanted a project that could combine hardware, software, and my love of retro computers into one. It seemed only natural to pick a project involving the Commodore PET.

I've always been fascinated by Commodore's prototypes and wonder what might have been if Commodore had decided to make a color PET fully

compatible to the original PET, rather than branching off with new incompatible systems.

So, my "ColourPET" project was born. I say "colour" because pictures I've seen of one color prototype spelled it this way, which makes sense since Commodore started as a Canadian company, with headquarters less than 30 minutes away from where I sit.

I decided to go "old school" and use only standard parts that were available back then. Later on, I could apply what I learned to more advanced projects, perhaps using CPLDs (complex programmable logic devices) or FPGAs (field programmable gate arrays). Luckily, many early vintage computers are quite hackable and the PET is no exception. I started looking at the schematics (**Figure 1**) and learning all I could about the hardware, as well as the firmware. It turns out that adding color to the PET is not as

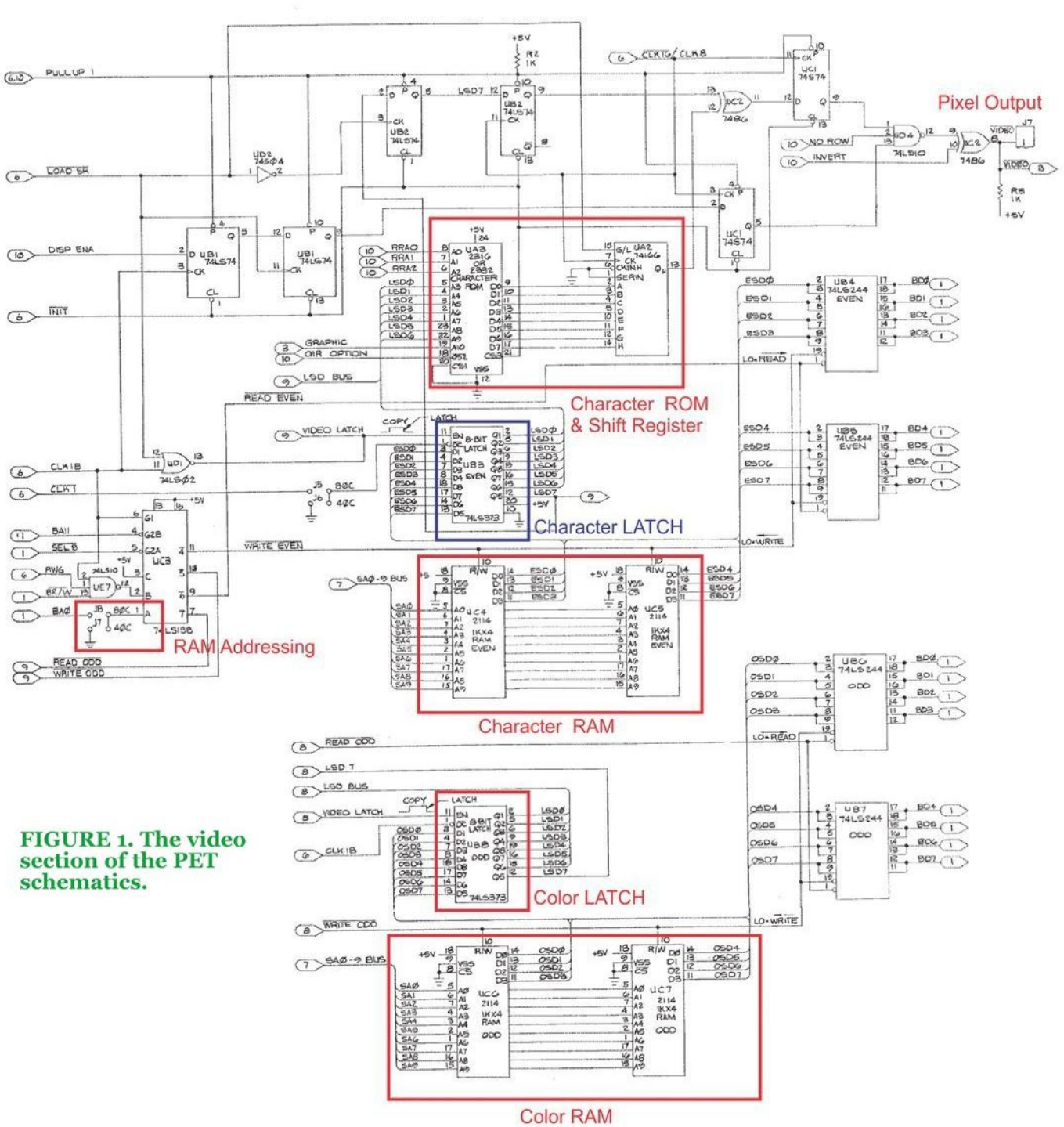


FIGURE 1. The video section of the PET schematics.

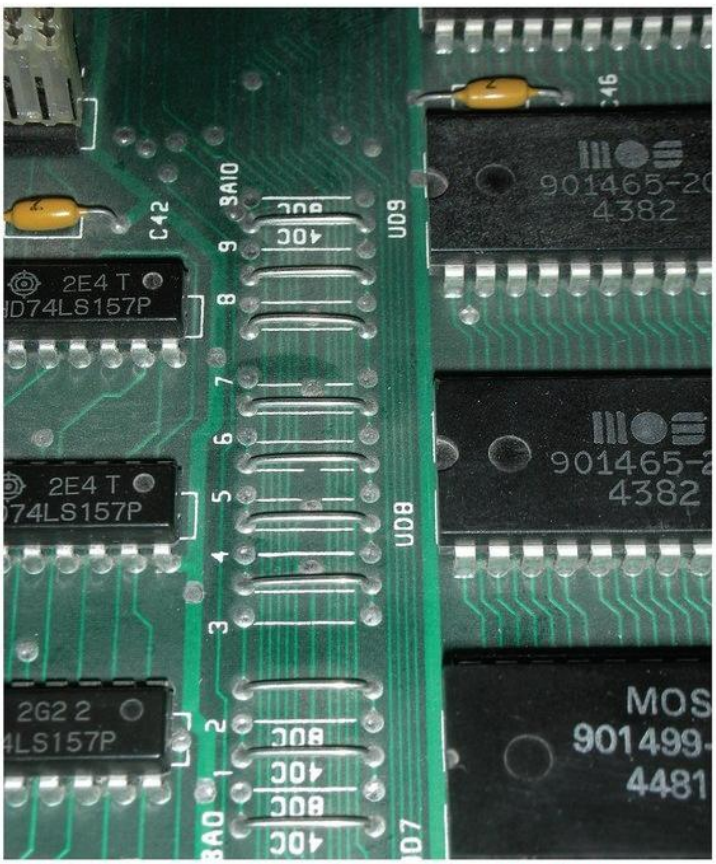


FIGURE 2. Jumper section set for 40 columns.

difficult as it might first appear, but in order to understand why, we need to look at how the computer is designed, keeping in mind limitations of technology at the time.

After the 2001 model, Commodore started using the MOS6545 cathode ray tube controller (CRTC for short). This chip is programmable and is responsible for producing the timing signals such as horizontal and vertical sync. The CRTC is fairly simple in that it accesses the video memory, but it does not actually produce the pixels we see on the screen. That job is left to additional circuitry.

The CPU, CRTC, and RAM are all clocked at 1 MHz, which was pretty typical back then. The CRTC reads one byte of video memory and latches it with a 74LS373 chip. That byte is sent to the character ROM along with lines from the CRTC, and then the appropriate character data is fetched and sent to a 74166 shift register clocked at 8 MHz, which puts out the actual pixels that we see. With monochrome video, our pixels are simply "on" or "off." A 40-column by 25-line screen needs 1,000 bytes, or just under 1 KB of RAM.

These first CRTC-based PETs (with 40-column screens) were not really business machines. Businesses preferred 80 columns to do their word processing, spreadsheets, and databases more efficiently. To create a new 80-column PET it would need double the video memory. That memory would normally have to be twice as fast, and that would mean the CPU would also need to be twice as fast due to the interleaved access method that the video system used. All of these would increase the cost because memory was quite expensive in the late '70s and early '80s ... fast memory even more so.

To keep costs down, Commodore came up with a clever solution which was to have two separate banks of memory. The CRTC could then read and latch two bytes at a time, effectively making a 16-bit wide video buss. This meant they could still use the same speed memory and same speed CPU. The shift register is now clocked at 16 MHz and sends the two bytes out one after the other, effectively doubling the pixels without changing the memory or CPU speed.

With this, Commodore re-designed the PET motherboard for 80 columns and the 8000 series machines were born. A short time later, they re-designed the motherboard yet again, to support 40 or 80 columns simply by setting a few jumpers on the board (**Figure 2**). This allowed them to sell either a "home" or "business" computer using one motherboard. It is this version — called the Universal Dynamic PET motherboard — we will use for this project.

Commodore unknowingly provided us with almost everything we need to add color to the PET. We have the CRTC chip, 2K of memory split into two banks (even and odd), a character generator and shift register, onboard jumpers for memory configuration, plus the right signals to control a monochrome CRT.

FIGURE 3. Early test with both monitors active.

When I first started studying the PET schematics, I tried to understand how the two video RAM banks worked and I realized that color systems do something similar. They have one area for storing the characters on screen and another to store the color. Both areas are read simultaneously just like in the 80-column PET. Ah-ha! To add color to the PET, all I needed to do was figure out how to use one bank of video memory for color information rather than character information.

We have 2 KB total of video RAM, so with two bytes for each character, we will only be able to display a 40-column color screen. Therefore, we need to set our motherboard jumpers to 40-column mode. Normally, the two video RAM banks are set so that one bank is for even memory addresses, and the other bank is set for odd addresses in a single memory range; in this case, appearing at $8000-$87FF in the memory space. We need to re-configure them so that one bank (the character RAM) appears at $8000-$83FF and the other bank (the color RAM) appears at $8400-$87FF. We do this by removing jumpers J7 and J8, then feeding the BA10 address line to the "A" input of the 47LS138 chip at UC3 pin 1.

FIGURE 4. Video timing software.

So, now the computer fetches both bytes and latches them as before, but now we need to use the color bytes to actually generate color. We have eight bits of data, so how do we use them? We could take all eight bits and use them to produce 2^8 = 256 colors for each character, then have a fixed background color such as black. We could output this to an analog RGB monitor by assigning some bits for RED, some for GREEN, and some for BLUE. We could then run those through a digital-to-analog converter (DAC) consisting of simple resistors.

Another option would be to split the eight bits into two groups, assigning four bits to the character's foreground color and four bits to the background color. With four color bits, we could use them as RGBI, giving 16 colors suitable for an RGBI digital monitor. I have designed circuits for both of these options, but for this project we will build the digital version as it's a little simpler.

The color data is available at the UB8 latch chip. We will tap into this chip, diverting the latched data to our circuit. Here's the crucial part of our plan: We are going to use the PET's normal monochrome video pixel signal as a toggle. When it is 0 (off), we use the background color; when it is 1 (on), we use the foreground color.

We can accomplish this with a 74LS157 quad multiplexer chip. We feed the four background bits to the A inputs and the four foreground bits into the B inputs. The monochrome pixel signal goes to the selector input and switches all four lines simultaneously. We now have a color PET!

We're not done yet. Where does this data go? We now output those four bits, representing RGBI to the monitor. Vintage monochrome and color monitors require differing sync signals and refresh rates. Our PET's internal monochrome monitor uses positive h- and v-syncs and runs at a 20 kHz refresh rate (**Figure 3**). We need to adjust these in order to connect to an RGBI monitor. These monitors require negative syncs and a 15 kHz refresh like early TV signals. A couple inverters will take care of the sync, but in order to adjust the refresh rate we will need to modify the timing of the CRTC chip.

The CRTC contains many registers that let you define the parameters of the video. When the PET powers on, it initializes the CRTC with routines in the EDITOR ROM. There are two tables in the ROM that define a "text" mode and a "graphic" mode. These tables will need to be modified to configure the signal for 15 kHz. During initial prototyping, I found that the refresh rates were similar enough that I could display a picture on both monitors at the same time, but the internal monitor was unstable and would blank out occasionally. To figure out what parameters were needed for the CRTC, I wrote a little program that let me adjust the chip registers in real time until I got a stable signal on the monitor (**Figure 4**).

By adding color, we find there are a few issues of concern. First, fetching data from the character ROM takes additional time. This has the effect of shifting the character and color data so that the first character on the screen

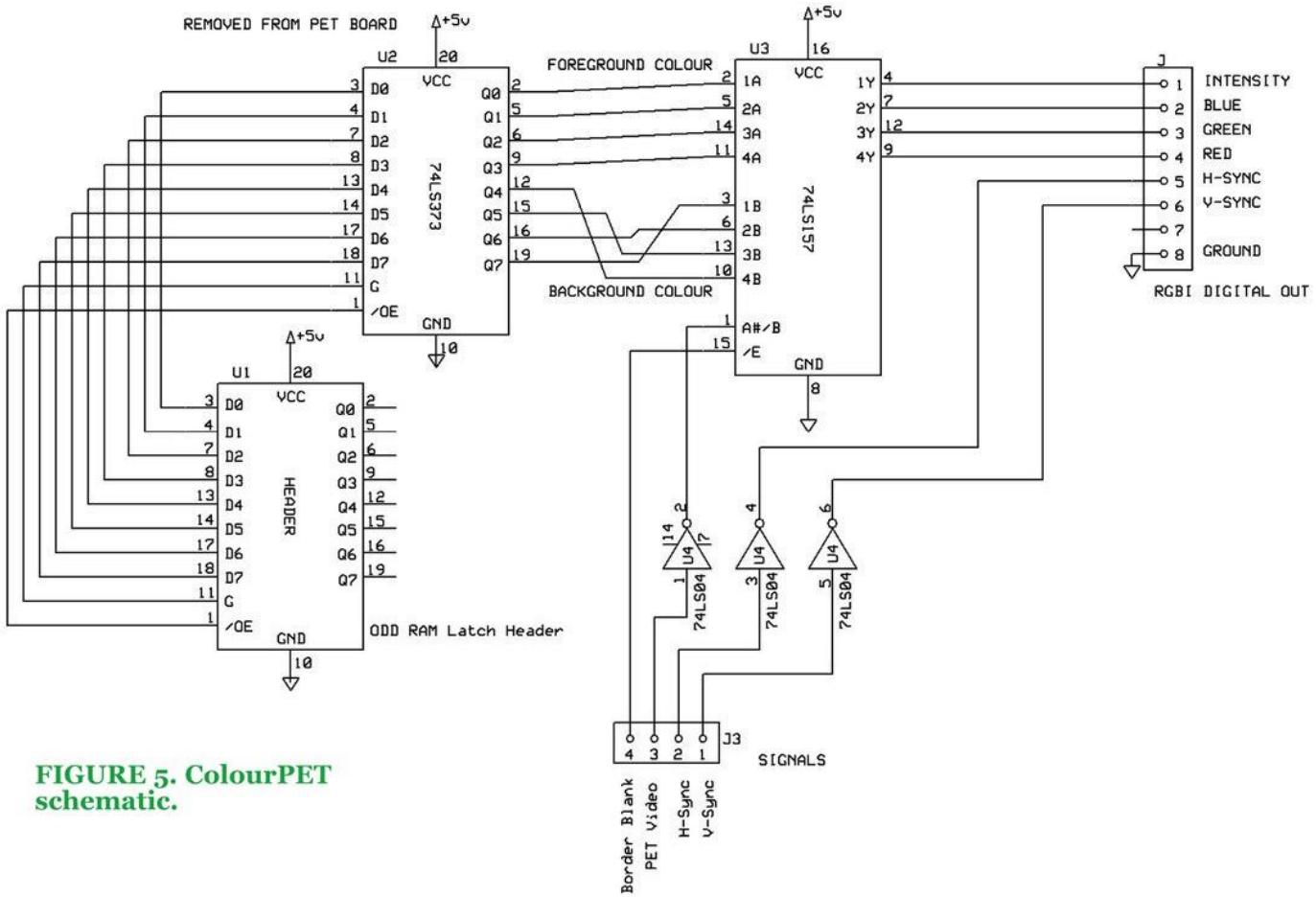


FIGURE 5. ColourPET schematic.

actually has the color from the second color RAM byte, the second from the third, and so on. It's not optimal but it works. The second problem is the border now shows phantom colors from characters on the screen. We will need to blank the border, which can be done by controlling the MUX output with the border blank signal at UC1 pin 6 on the motherboard.

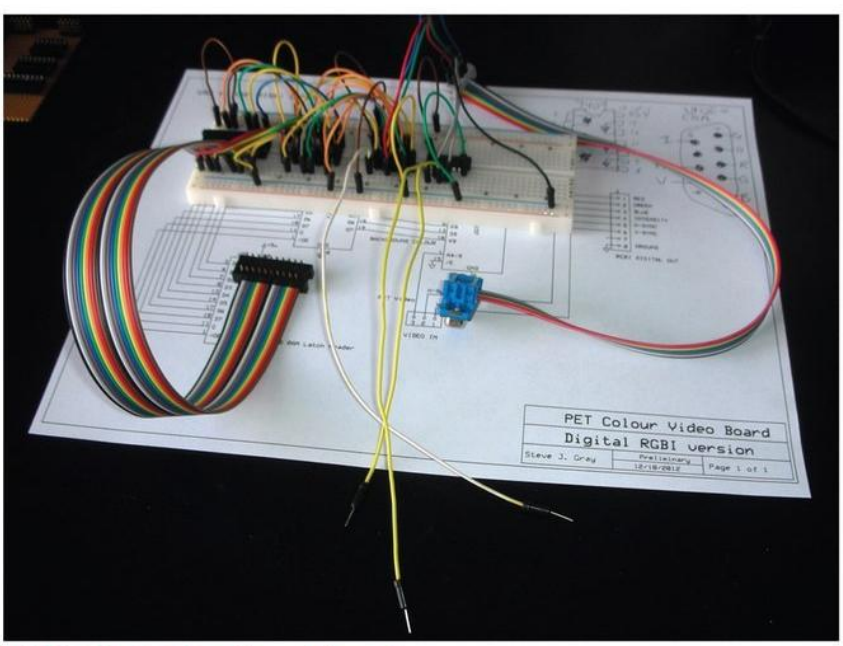


FIGURE 6. First breadboard.

All of this hardware would be useless without code to support it. Things like clearing the screen, scrolling, inserting, and deleting characters need to account for the color RAM. Just printing characters needs to write to the color RAM, plus we need some way to change the foreground and background colors. We need to initialize the CRTC controller properly, and we probably want to change the sign-on message. That's a lot.

Early on, I started by patching things manually and burning a new EPROM, but it soon became apparent that this method was very time-consuming and not very flexible.

The PET firmware consists of a KERNEL — much like a PC's BIOS — plus BASIC and, as mentioned earlier, the EDITOR ROM. The EDITOR ROM is responsible for all keyboard decoding, interrupt handling for such things as

the clock, and especially for screen handling. All these functions were grouped so that Commodore could customize the machine for different markets and different hardware variations; for example, European DIN keyboards, 50 Hz clock timing, and, of course, 40- or 80-column screens.

I have disassembled the EDITOR ROM code and created commented 6502 assembly language which allows us to make any changes or additions needed, or build an EDITOR ROM for any combination of PET features/options we like such as keyboard layout, screen size, or even adding advanced screen editing functions like escape sequences. I have made this code available on GitHub for anyone to look at or contribute to.

The code is formatted for the multi-platform ACME assembler. The main file is called "edit.asm." At the top are several settings that can be changed to suit your system. You will need to set "COLOURPET=1" to enable the color features, "COLUMNS=40," and "COLOURVER=0." Lastly, you need to set KEYBOARD and REFRESH properly depending on your PET model.

After assembling the source, you will get a 6502 binary file that can be burned to 2532 or 2732 EPROM which will replace the ROM in location UD7. Alternately, you can download some ready-to-go binaries from my project web page.

While developing code for the ColourPET, I discovered that Commodore put a nice jump table at the start of the EDITOR ROM to all the routines, and then didn't use them. Several KERNAL routines jumped into the middle of the ROM, which made adding code very difficult. The standard EDITOR ROM is 2 KB in size and is almost full, but luckily, the PET supports a 4K EDITOR ROM, giving us a lot of space for additional code.

Our ColourPET code will go into the upper 2K. What I ended up doing was moving blocks of code, then using *JSR* (jump to subroutine in 6502 assembler speak) to the new routines with my own patched code. I then added *!FILL* directives to pad out the code where needed to ensure that those fixed entry points did not move.

I built the circuit (**Figure 5**) first on a solderless breadboard (**Figures 6** and **7**) and then later on a protoboard using point-to-point wiring. I chose to mount the board on the video connector with a ribbon cable going to UB8 to allow easy inspection of the board near the front of the machine (**Figure 8**). You could also build

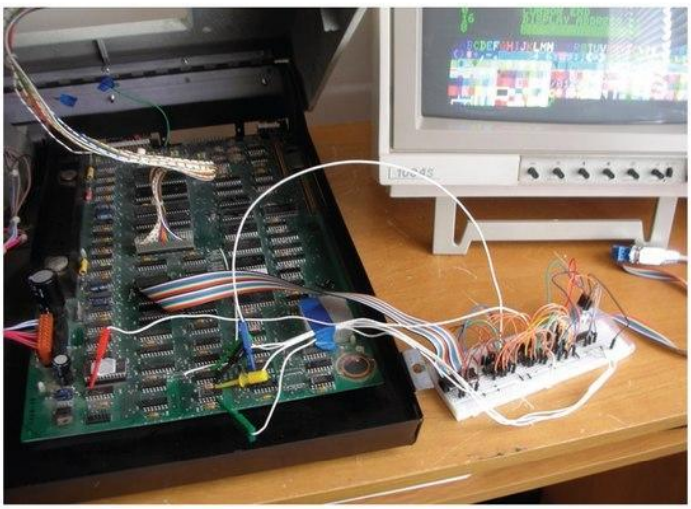


FIGURE 7. Breadboard connected to the PET.

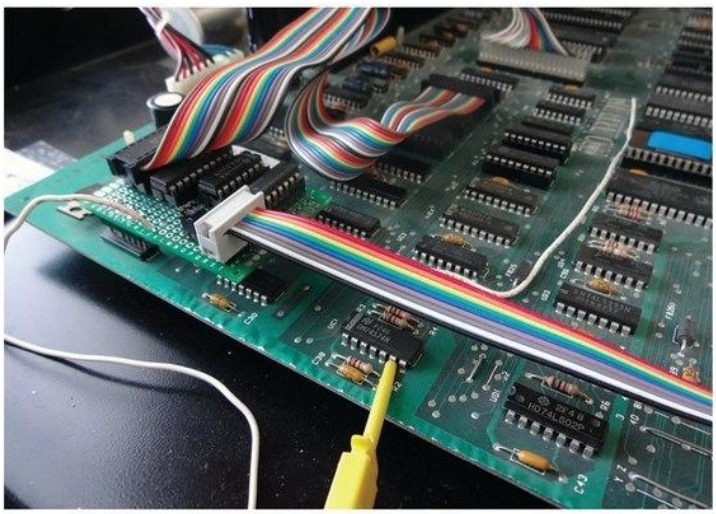


FIGURE 8. Final P2P board mounted on video connector, ribbon cable to latch (top), white wire to remap RAM, and yellow clip to border blank signal.

the circuit so that it mounts on UB8 with three wires going out to the video connector and one to the border blank signal.

On my prototype, I removed the original monitor and connected my circuit to a Commodore 1084-S monitor, which supports both RGBI digital and RGB analog signals. Eventually, I want to try mounting a color monitor in the original PET monitor housing so that it looks like Commodore might have actually built a real color PET.

I initially started with a 4032 PET and the solderless breadboard. If you have a 4000 series PET with the correct motherboard, you will need to add the extra missing memory and support chips. My second prototype with a point-to-point board was built into an 8032 PET. If you have an 8000 series PET, those chips are already installed. These systems normally come with different keyboards, and I wanted to have the version with the ESC key so I could do development on the EDITOR ROM.

The project presented here is just a starting point. Many improvements can be made. I have started designing the next generation ColourPET board (**Figure 9**). We could combine the digital and analog methods, or we could add programmable borders and background

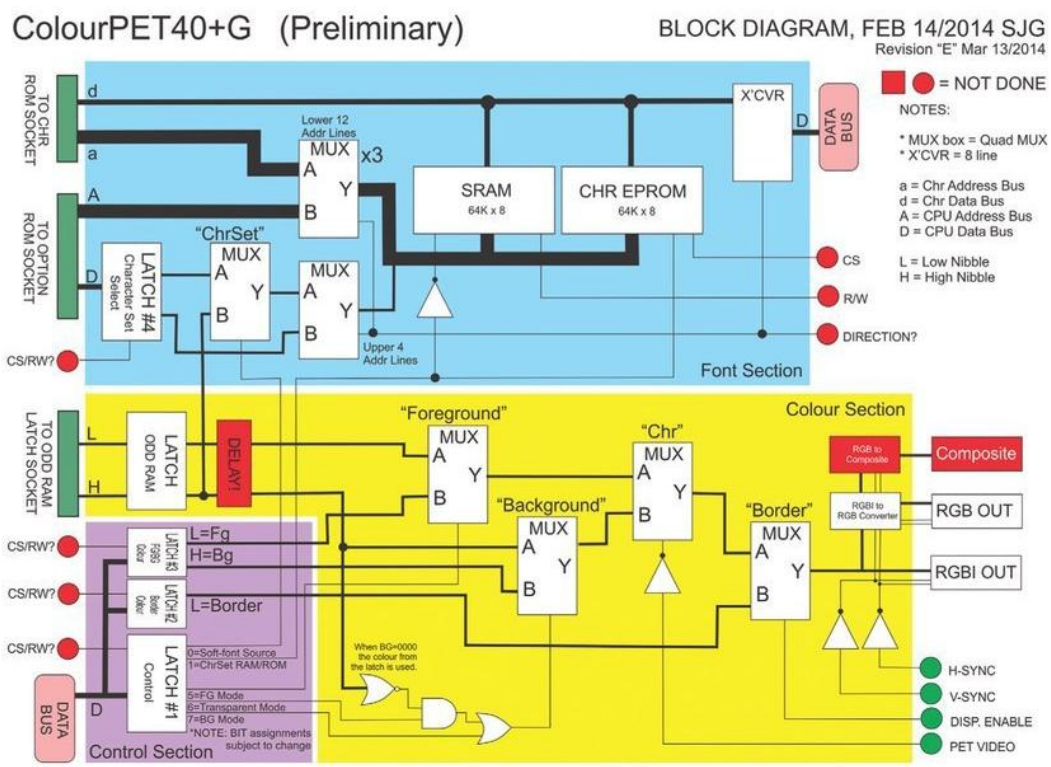


FIGURE 9. Design for the next generation ColourPET board.

Resources

Personal Webpage
www.6502.org/users/sjgray/index.html

ColourPET Webpage
www.6502.org/users/sjgray/projects/colourpet/index.html

Editor ROM Webpage
www.6502.org/users/sjgray/projects/editrom/index.html

Editor ROM Source
https://github.com/sjgray/cbm-edit-rom

colors, or even transparency. Taken further, we could add additional RAM to allow 80 columns, or even add graphics or sound capabilities to give the PET features rivaling later machines like the C64. Check my web page (see **Resources**) for more ideas, schematics, and support files. Don't you think it's time to start your own PET project? **NV**

Steve J. Gray

- First computer OSI C4P, 1981.
- Commodore collector since 1999.
- Creator of MP3+G format, 1998.
- Co-founder Tyrannosoft, specializing in developing Karaoke software, 2000.
- Member of TPUG 2007-present. Longest running Commodore User's Group in the world.

Beyond the Arduino 6

To the Power of I²C

The components we've included in our projects so far have been relatively simple. This month, we raise the bar and connect our AVR microcontroller to modules with a little more intelligence, using the I²C protocol.

There is Intelligent Life Out There

After five months of sitting on a breadboard with mere LEDs and switches for company, your ATmega328P microcontroller has likely started to develop a deep longing for intelligent conversation. Sure, it has spoken with your PC over a USB-serial connection, but that is still the only component with any form of intelligence on the breadboard. You can reassure your MCU that this month, things are about to change!

From your experience with the Arduino environment, you'll be familiar with some of the intelligent components we can add to our projects: a host of digital sensors; real time clocks (RTCs); EEPROM modules; modules that enable wireless/Bluetooth/LAN communication; GPS modules ... the list is a very long (and exciting) one – and one that can inflict long-lasting damage on your credit card!

As these modules can provide some fairly complex functionality, it follows that there has to be some degree of complexity in the way we communicate with them – a simple high or low voltage on a microcontroller pin simply won't cut it any longer.

The two most popular protocols used to communicate with these kinds of modules are: SPI (Serial Peripheral Interface); and I²C (Inter-Integrated Circuit). These two protocols achieve the same basic result – communication between modules – but as with most things have their own strengths and weaknesses.

At a summarized level, SPI allows faster and simpler communication, but needs four wires (plus GND) to achieve this. I²C is slower, but only requires two wires (plus GND) *and* allows you to daisy-chain multiple modules (up to 128 of them) on the same pair of wires. The I²C simplicity at a hardware level is, however, a trade-off on the software side – it is a more complicated protocol.

Making a Choice

So, how do you decide whether to use SPI or I²C on your projects – assuming, of course, that both versions of your module are available (some manufacturers are kind to us and do offer both versions)? The simple answer is there *isn't* a simple answer – your choice will depend on what constraints you face in your particular project; what trade-offs you're prepared to make; and what aspects you aren't prepared to compromise on.

The first stand-alone AVR project I completed that had any level of complexity was the irrigation controller I've mentioned in previous articles. In addition to the microcontroller itself, I had an EEPROM chip, an RTC, and an LCD display (as well as various buttons, LEDs, relays, and a rotary encoder). My constraints were budget and the number of I/O pins I had on my MCU (I was using the same ATmega328P we're using here). I had no requirement for response times or volume/speed of data transfer, so I could afford to compromise on that.

The decision I ended up making at the time was to manage my budget by avoiding a serial LCD and instead using a shift register to address the LCD in a "traditional" way. I then opted for an I²C EEPROM and I²C real time clock as I could use the same two pins to communicate with both – I had no need for the SPI speed, and could not spare the I/O pins I would have needed to use SPI versions of the modules. A simple example perhaps, but nevertheless it illustrates the point.

Diving into Detail on I²C

If you've been following this series, you'll know that I like to get into a little theory before we apply power to a project. Let's take a quick look under the hood of I²C.

By Andrew Retallack
www.crash-bang.com

The ATmega328 datasheet gives some quite detailed information on the protocol — we're not going to get into that level of detail here since the microcontroller takes care of much of the behind-the-scenes complexity for us. A quick note: The datasheet refers to I²C as **TWI**, or Two Wire Interface. This was to avoid "complications" with the trademark that Phillips (now NXP) registered on the I²C protocol; however, TWI is completely I²C compatible.

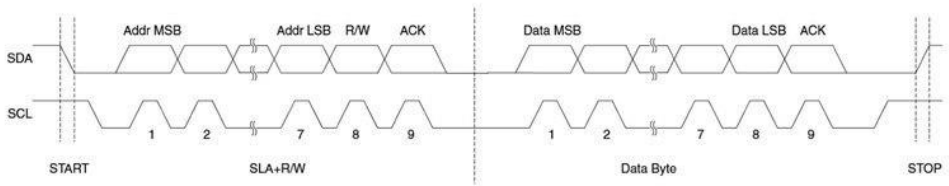


Figure 1: Typical I²C timing diagram.

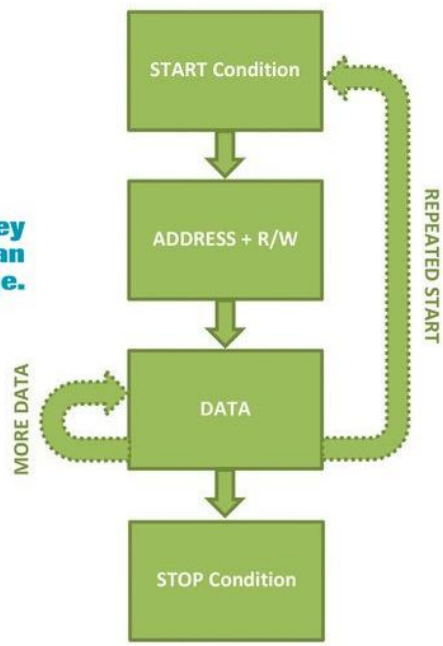


Figure 2: Key elements in an I²C exchange.

Get Connected

Let's start with the physical connections. In addition to a common ground, I²C only needs two connections between the modules wanting to communicate: Clock (called SCL — Serial CLock) and Data (called SDA — Serial DAta). I²C differs from the serial UART communication we looked at a couple of months ago in that the UART operated at a specific speed (baud rate). I²C is more flexible in that it doesn't need to agree to a speed between the modules up front, but operates at the speed that the clock signal dictates.

One bit of data is transferred each time the clock signal pulses — the clock signal is a bit like a metronome, setting the pace for the module's communication. The advantage of this type of communication is that the modules don't need accurate crystals or oscillators — a microcontroller can communicate with all sorts of (relatively simple) modules, without worrying about whether they're going to stay synchronized.

The data line operates much as we'd expect — as a series of voltages: high levels (for a "1" bit) and low levels (for a "0" bit). String the bits together in time with the clock pulses, and you're soon transferring full bytes.

Remember Your Pull-Ups

The signals for I²C are referred to as open-drain. In other words, their default state is high and they need to be pulled low. Let's digress to illustrate this: If you were to connect an LED with the cathode (negative terminal) to your microcontroller and the anode to your power rail, you would be operating on an open-drain setup. For the LED to be off, you'd have to ensure the pin on your microcontroller was high (usually by using a pull-up resistor). To turn the LED on, you'd pull the pin low (i.e., to GND), allowing current to flow.

I²C operates on a similar principle. The lines are held high by pull-up resistors, and then pulled low by the modules wanting to communicate. When the modules stop pulling the lines low, they "bounce" back up to a high state. So, remember, I²C must always be accompanied by pull-up resistors on the SDA and SCL lines. The datasheets will usually give you an indication of the value of the resistors, but a value from 2K to 10K usually does the trick.

Building Up the Protocol

I mentioned earlier that I²C has a slightly more complicated protocol as a trade-off for the simplicity of the physical connection. While you don't need to understand how the protocol works at a physical level as signals transition between high and low (timing diagrams like those in **Figure 1** can get a bit overwhelming), you do need to know the protocol for having a conversation. **Figure 2** summarizes the key elements of an exchange, detailed next.

The Start Condition

Every communication needs to begin with a start "condition" which occurs when the SDA line is pulled low before the SCL has started pulsing. This warns the connected modules that something is coming — a bit like clearing your throat before you address a group of people.

Rock Around the RT Clock

Real Time Clocks are wonderful additions to projects. Sure, they do what they say (act like a clock), but as most microcontrollers have no sense of human time or timescales, a clock can be far more useful than simply telling the time.

Here are some features that you could expect from RTCs:

- **Set and Tell the Time**: Yes, it goes without saying! You'll need to get used to the specific registers that are needed to set and read the different elements of time — including configurations like time format (24 or 12 hour) and day of week. A good RTC is also able to handle leap years for you.
- **Alarms**: These are great, as they allow you to schedule specific actions to take place at specific times without you having to write the code to keep checking "is it time yet?" Alarms can often be set to go on a specific year/month/day/hour/minute/second — so you can get really specific about when events take place. Perhaps you only want that project to come to life at midnight on Halloween. You would normally use an alarm to trigger an interrupt on a specific pin on your microcontroller, and have the microcontroller take it from there.
- **Backup Battery**: The clock on my home oven drives me crazy — each time the power cuts (a common occurrence where I live), I need to set the time again. Fortunately, your project can have a backup battery (a tiny coin cell) so that you don't face the same frustrations. Some RTCs will even tell you when the power cut, so you can compensate for events that should have occurred during the outage.

All of the above are controlled through registers on your RTC — registers that you access through the I²C interface, and registers that are detailed in the datasheets.

The Address

Next up, you need to let the connected modules know which one you're going to communicate with, in the same way as you'd call a person out of a group by name. Each I²C module has a unique address (or address range) that is assigned by NXP — seven bits in total. Some modules allow you to set the last few address bits by connecting them to 5V (resulting in a 1 bit) or to GND (resulting in a 0 bit). We'll see this in operation in our project later. The address byte is made up of the seven-bit module address, along with an eighth R/W bit that specifies whether you want to write to (a 0) or read from (a 1) the module.

Address Acknowledged

For each byte sent, the receiver of the byte needs to acknowledge that it has been received with a ninth bit. This is called an "ACK" and is a 0 over the SDA line. If the receiver doesn't acknowledge (called "NACK"), then the transmission is deemed not to have been received. You'd need to decide in your software how to handle this. For example, you may want to retry or terminate the communication for a period.

Data Transfer

After the address has been acknowledged, you're ready to start communication. Remember that the direction of the communication depends on the value of the last bit in the address byte.

If you want to change the direction of the data, you can simply re-issue another start condition (called a repeated start) and resend the address byte with a different data direction bit value.

Stop Condition

Finally, to end a transmission, a stop "condition" needs to be sent. This is the opposite of a start condition, in that the SCL is first released high (the clock is stopped), and then the SDA is released back to a high state.

Using the Protocol

Understanding how the protocol works is not that straightforward. Thankfully, the microcontroller takes care of much of the low-level detail by means of its two wire serial interface — you don't need to know which lines to hold high or low at what stage. The Atmel datasheet actually does a pretty good (and in-depth) job of working through the TWI module if you're interested in the technicalities. Okay, so we've had a quick look at *how* modules communicate, but not yet *what* they communicate. That is a whole other challenge!

I²C modules (and SPI modules for that matter) are usually able to perform more than one type of function and often allow you to configure their behavior in certain ways. For example, an EEPROM module may only allow you to write to a specific address or to read from a specific address. A real time clock, on the other hand, could allow you to set the time, read the time, configure alarms, specify the format that the time is in, and more. Each module behaves very differently, and therefore needs to be communicated with differently. Brace yourself, because we need ...

... More Datasheets, More Registers

Each module you'll be using over I²C will behave differently. It would be impossible to standardize an interface because the range of functionality varies so widely. As you could guess from our interactions with the ATmega328P, this means a datasheet to detail the interface and using registers to access the functionality. It would be impossible for an article — or dare I say it, even a book — to cover all the I²C modules out there; so let's work through a project that interfaces with an I²C module or two to get a better understanding of the protocol. I'm a big believer that the most effective learning happens by

doing. So, let's get "doing."

Get Doing: Feel the Heat

This month, we're going to work on our most complex project yet: a temperature logger. Each month, we've tried to build on the learning from previous months, so that we work towards more complex and more useful projects.

Anatomy of a Temperature Logger

We'll be building our project from four key elements: the microcontroller; a temperature sensor; an EEPROM module to store readings; and an RTC so we know what time we took the readings. Finally, we'll use our faithful serial-USB converter to allow us to keep the user informed through a serial monitor on their PC.

The microcontroller is, of course, the brains. It pulls all the components together:

- Sets the time on the RTC
- Reads the temperature on the temperature sensor using the ADC (analog-to-digital converter)
- Reads the time off the RTC
- Stores the time and temperature reading on an EEPROM chip
- Manages serial communication with your PC

We've sneaked a couple of exciting new components in — an EEPROM IC and an RTC module. The EEPROM chip is fairly straightforward to use — you are simply reading from or writing to specific addresses on the chip. The RTC is a little more complex, so I've gone into more detail in the sidebar, "Rock around the RT Clock."

Process Flow

Before we get into the nuts and bolts of how everything works, take a few minutes to look over the flowchart in **Figure 3** in order to understand the process flow of the logger. You should recognize some of the approach from previous articles in this series. Let me flesh out the flowchart a little.

When the logger is powered on, a standard initialization needs to take place. This time, we have a host of interfaces and peripherals to initialize. You should be familiar with all but the I²C interface.

We'll be using a serial (UART) interface again to allow us to configure the logger, as well as a means to extract the logged temperatures — for example, if we wanted to graph them. We need to be able to set the time, clear the log, and print the log using a UART menu. You could add additional functionality yourself — for example, to change the logging frequency — but for now we'll leave that out. Just as we did before, we'll use interrupts to detect an incoming command over the UART.

Once the logger is running in its infinite loop, it will continually check whether it is time to log another temperature reading. We are not using a very efficient mechanism for this — simply a delay and a counter. There are better ways to do this using timers (which we'll discuss in a later article) or even using built-in alarm functionality on the RTC — but for simplicity, we're going to sacrifice efficiency. When it's time to log a temperature, the process is fairly straightforward:

- Read the temperature from the sensor using the ADC (exactly as we did in the article in the May edition of *Nuts & Volts*)
- Read the time from the RTC
- Write the log entry to the EEPROM
- Write the time and temperature over the UART (just so we can see things are happening!)
- Reset the counter and start over

Registers, Registers, Registers

Yes, I'm afraid it's time to peek at the registers on the ATmega328P that manage the I²C functionality. Open up your datasheet to section 22 TWI, and if you're feeling brave, read through some of the details in this section. If you're not feeling brave, don't worry. I don't think that I've read the entire section myself! Refer to **Figure 4** for a look at the key registers we'll be dealing with here, and you'll see that there are (thankfully) only a few that we need for

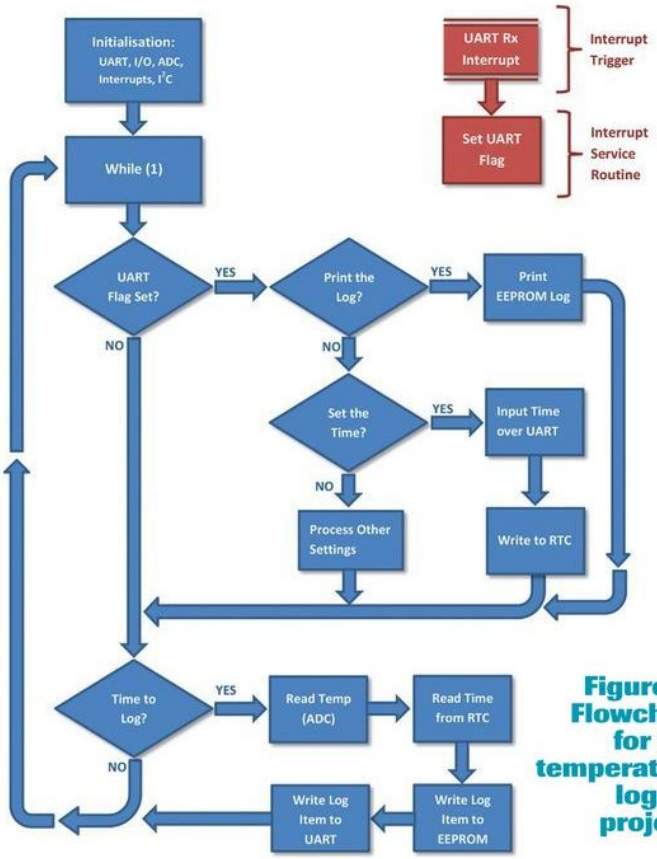


Figure 3: Flowchart for the temperature logger project.

TWSR – TWI Status Register

Bit	7	6	5	4	3	2	1	0
Function	TWS7	TWS6	TWS5	TWS4	TWS3	-	TWPS1	TWPS0

TWI Communication Status — TWI Prescaler

TWCR – TWI Control Register

Bit	7	6	5	4	3	2	1	0
Function	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE

Interrupt Flag — START Condition — STOP Condition — Enable TWI Comms

TWDR – TWI Data Register

Bit	7	6	5	4	3	2	1	0
Function	TWD7	TWD6	TWD5	TWD4	TWD3	TWD2	TWD1	TWD0

The Data

TWBR – TWI Bitrate Register

Bit	7	6	5	4	3	2	1	0
Function	TWBR7	TWBR6	TWBR5	TWBR4	TWBR3	TWBR2	TWBR1	TWBR0

The Bitrate

Figure 4: The I²C registers used in our project.

now: the Status register; the Bit Rate register; the Control register; and the Data register. We'll look at these in more detail once we get coding.

Laying Components on the Breadboard

Figures 5 and **6** show the schematic and partial build for our project — far more complex than our first five projects! I've excluded the components needed to get the microcontroller working on a breadboard (as in the past articles) in order to allow us to focus more easily on these specific components. Grab your datasheets and let's quickly run through how they're connected!

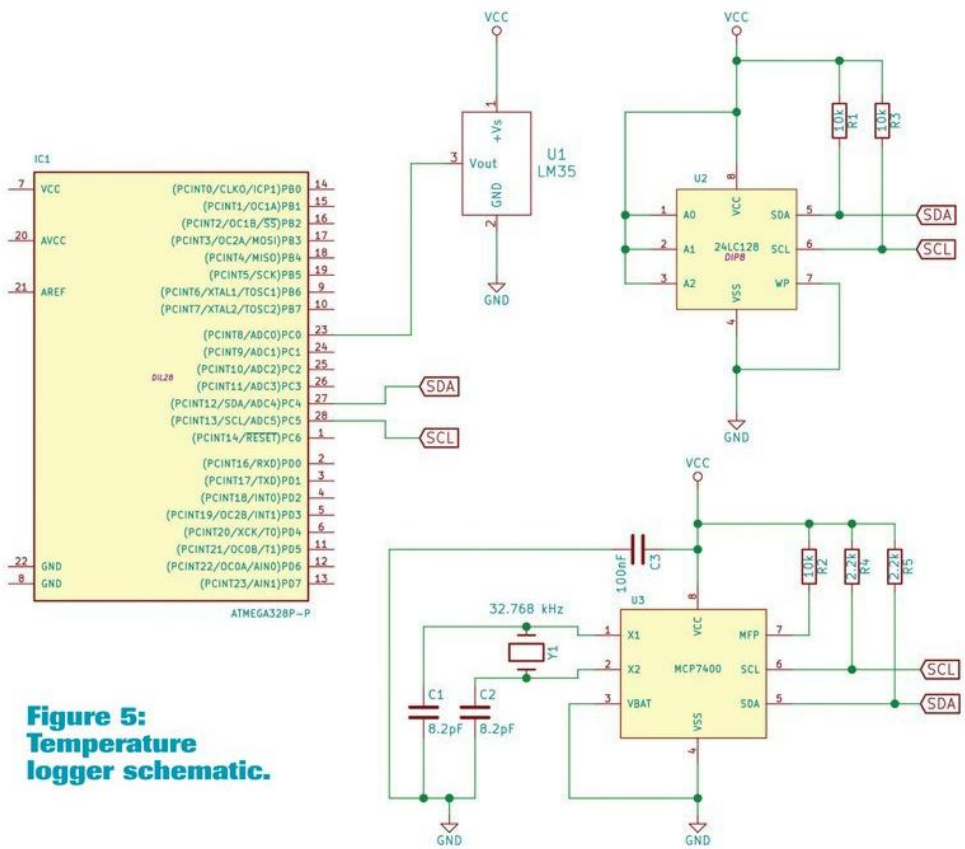


Figure 5: Temperature logger schematic.

U2: 24LC128

This is the EEPROM chip: Microchip's 24LC128 (refer to Resources for a link to the datasheet). It has 128 kilobits of storage, and operates between 2.5V and 5.5V. It's available in various packages including a DIP (so is breadboard friendly).

Taking a look at the connections, you'll immediately notice that the **SDA** and **SCL** lines have 10K pull-up resistors — we mentioned these pull-ups earlier. These pins are connected to pins 27 and 28 on the ATmega328P — the microcontroller's own SDA and SCL lines.

Pin 7, **WP**, is tied to GND — this is the write-protect pin, and is disabled when connected to GND. I once spent hours troubleshooting a project, only to find that I'd misread the datasheet and connected the write-protect pin to VCC instead of VSS (GND).

Finally, there are three pins labelled **A0** to **A2**. This is a nice touch, and enables you to set the I²C address for the chip. Remember that the I²C protocol requires you to specify which component you're talking to by using its address. These pins allow you to connect more than one 24LC128 chip by giving each their own address. The base address for the 24LC128 is "1010*xyz*" where ***x*** is the value on pin A2, ***y*** is A1, and ***z*** is A0 (refer to the datasheet for more details; "Section 5.0: Device Addressing"). In our project, I've connected all three pins to VCC, so the address is 1010111. Remember, it's only seven bits in length, as the eighth bit is the read(1)/write(0) bit.

U3: MCP79400

This is the RTC chip: Microchip's MCP79400 (see Resources). Apart from the basic timekeeping (date and time), it also has a backup battery and two alarms. I haven't included the backup battery on the schematic — opting for a basic configuration — but once you've mastered this project, it'll be easy for you to include it. The one feature I really like about the MCP79400 is the wide voltage range it can operate over: from 1.8V up to 5.5V. It doesn't really do much for us now

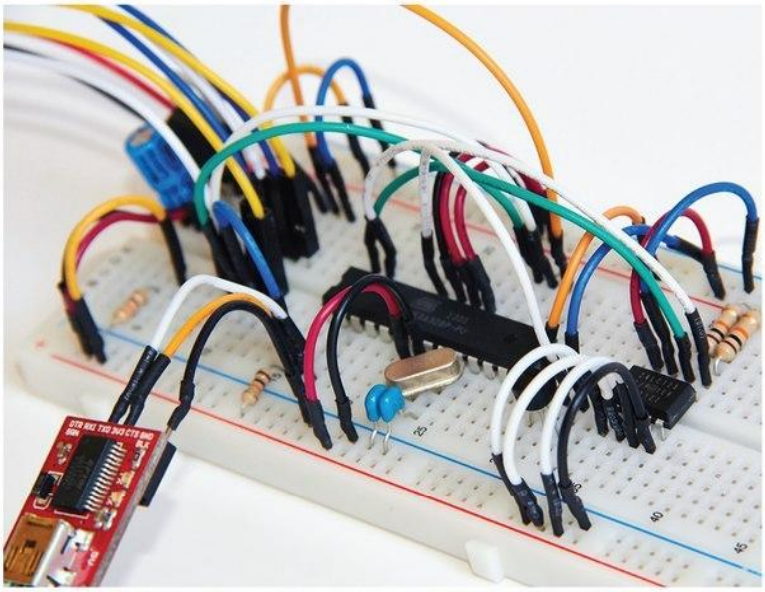


Figure 6: Breadboard build: the EEPROM module.

with this project, but when we start getting into low power projects this becomes critical.

One of the disadvantages to using the MCP79400 is that it doesn't come in a nice breadboard-friendly DIP package – meaning that you need a breakout board. This was a compromise I was prepared to make based on a combination of the price and the voltage range – not many other RTCs offer the range at the price (for example, the very popular Maxim DS1307 only operates at 5V). Additionally, a breakout board for an RTC is a pretty useful thing – you'll see that the RTC needs a number of components which can be nicely wrapped up into a breakout board.

Onto the connections! Again, the **SDA** and **SCL** lines have pull-up resistors; although this time, they're 2.2K resistors. If you take a look at the datasheet, you'll see that a range of pull-ups will work, although do affect the possible operating speed. However, this is a longer discussion for another time!

A watch crystal/tuning fork crystal (i.e., rated at 32.768 kHz) is connected to **X1** and **X2**. This is needed to provide accurate timekeeping, along with a couple of 8.2 pF capacitors – similar to the configuration for the crystal on your main microcontroller. **VBAT** is for the backup battery, but here is attached to GND as we don't have one connected. Finally, the **MFP** (multi-function pin) is kept high with a 10K pull-up resistor – this pin can be used to indicate when alarms are triggered, to output a square wave (let's say you want an LED to flash as a second indicator), or even be an additional GPIO.

Whew! That was a quick overview!

Changing Gears

We've covered some of the theory and the physical layout. Now, you can put down your breadboard and fire up Atmel Studio. Let's get coding!

Project Structure

Download the project from the article link and open it up in Atmel Studio. Then, take a look at the solution explorer tree; you'll see something like **Figure 7**.

You probably noticed that there are more files than before: I2C.c, I2C.h, UART.c, and so on. As this project is getting large, I've separated out various groupings of functionality into separate files – this makes our lives a great deal easier. Look at the sidebar, "Separate Your Code, Then Include It" for some more detail. Take a quick look

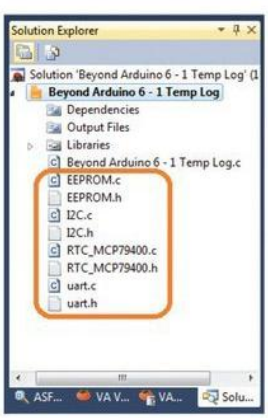


Figure 7: Solution Explorer: Additional library files.

Separate Your Code, Then Include It

Just as it is good coding practice to separate your code out into modular functions, it is good practice to group similar functions into distinct files. For example, in this article's project, there are code files containing functions relating to UART communication, accessing the EEPROM, communicating over I²C, etc. These files have been given meaningful names that reflect their purpose.

I'd be doing a poor job if I tried to get into a discussion on the best coding practices for your projects — this is a whole field in itself and not one that I can claim to be an expert in; a search on the Internet will throw up a number of academic and practical books on the subject. However, for our purposes, I'm comfortable that we'll see the benefits as enthusiasts. Separate files mean that it's easier to find specific functions without wading through a single lengthy program file. By testing a file completely and making sure it can stand alone, you're able to use this in other projects without re-testing. You know that your (for example) UART functions are sound and can be included in any future projects for "plug-in" functionality.

You'll have noticed that there are two files: a header file (e.g., "UART.h") and a source file (e.g., "UART.c"). The header file should not include any functioning code — only definitions, macros, function prototypes, etc. The source file needs to "include" the header file — for example, "UART.c" has a statement at the start:

```
#include "uart.h"
```

This brings the header file and its definitions into the source code file.

Additionally, all the header files need to be included in the main code file. Look at the start of the main "c" file for the project and you'll see all the "#include" statements there.

We'll probably touch on header files at a later stage, but this will give you enough background to get started.

```
uint8_t EEPROM_write(uint16_t deviceAddress,
uint16_t memoryAddress, uint8_t data)
{
    uint8_t result;
    uint8_t returnResult = 0;

    //Send START Condition
    result = I2C_sendStart();

    //Send the device Address with WRITE
    result = I2C_send(deviceAddress|TW_WRITE);

    //Send Memory Location Address to write to
    //(High)
    result = I2C_send(memoryAddress >> 8);
    //Address High

    //Send Memory Location Address to write to
    //(Low)
    result = I2C_send((uint8_t)memoryAddress);
    //Address Low

    //Write Data to EEPROM
    result = I2C_send(data);    //Data

    //Send STOP Condition
    I2C_sendStop();

    _delay_ms(10);

    returnResult = 1;
    return returnResult;
}
```

Listing 1. Function to write a byte to EEPROM.

Resources

Author's website:
www.crash-bang.com

Microchip's 24LC128:
www.microchip.com/wwwproducts/Devices.aspx?product=24LC128

Microchip's RTC MCP79400:
www.microchip.com/wwwproducts/Devices.aspx?product=MCP79400

Details on the Toadstool:
www.crash-bang.com/projects/toadstool/

inside each of these files to get a feel for what functionality they include. The sample project is pretty well commented and aligns with the flowchart, so it should be relatively easy to navigate through — and allow us to focus on the specifics of I²C.

The Five I²C Functions

Yes, that's right — there are only five main I²C functions that we need to build. Once those are in place, then we can communicate with pretty much any I²C module. I've placed these in the "I2C.c" and "I2C.h" files.

Initialize the I²C Interface

This is really easy, as the code in **Listing 1** shows. We simply set a prescaler (something you'll remember from our ADC examples) in order to slow our system clock down to a speed that the I²C interface can use, and then set the speed (*bitrate*) that we want to communicate at — usually 100 kHz to 400 kHz, depending on the module and the pull-up resistor values:

```
void I2C_init(uint8_t I2C_kHz)
{
  uint8_t bitRate;
  uint16_t I2C_Hz = (unsigned long)I2C_kHz *
1000UL;  //Convert kHz to Hz

  //Calculate bit-rate
  bitRate = (F_CPU / (I2C_Hz * 2 *
I2C_PRESCALER)) - (16 / (2 * I2C_PRESCALER) );

  TWSR &= ~((1<<TWPS0)|(1<<TWPS1));
//Clear the pre-scaler
  TWSR |= (1<<TWPS0);                    //Set
the new pre-scaler to 4

  TWBR = bitRate;    //Set the bit rate

}
```

If you look at the I2C.h file, you'll see that *I2C_PRESCALER* is defined as 4 (as we are running at a clock speed of 16 MHz). The bit rate is calculated using the rather long formula from the datasheet — and relies on the value of the prescaler. So, to set the prescaler we set the **TWPSO** and **TWPS1** bits of the TWI Status Register (**TWSR**) using Table 22-7 from the datasheet. The *bitrate* is set in the TWI Bitrate Register (**TWBR**). That's about it; let's start.

Send a Start Condition

Remember that each exchange begins with a "Start Condition" — pretty logical, I guess! This is a simple call:

```
TWCR = (1<<TWINT)|(1<<TWSTA)|(1<<TWEN);
```

We work in the TWI Control Register (**TWCR**) and:

- Clear the Interrupt flag (even though we aren't using interrupts right now)
- Set the Start bit
- Enable TWI (I²C), which sends the communication

There is one small thing, though. As with some of the other functionality we've used in the past, we need to make sure the task has completed. For this, we need to wait until the Interrupt flag is set:

```
while (!(TWCR & (1<<TWINT)) );
```

We could use this flag to drive an interrupt-handler, but for now we'll keep things simple.

Send the Data

The I²C interface doesn't care what data you send — in other words, it doesn't differentiate between the initial control byte (containing the address), the address of a register on the I²C modules, or data you're putting into a register on the remote module. As a result, it's really straightforward to transmit data:

```
TWDR = Data;  //Set the Data to send
TWCR = (1<<TWINT) | (1<<TWEN);  //Clear
interrupt flag and enable TWI
```

We load the byte we want to send into the **TWDR** register, clear interrupts, and enable TWI in the **TWCR** register. Of course, we again need to wait around until the data is sent. To make it easier on myself, I created a simple function that I call after my I²C communications — you'll see it in the project code:

```
void I2C_waitComplete(void)
{
       while (!(TWCR & (1<<TWINT)) );
}
```

Read Data

Often, if we send data, we expect something back in return — for example, we could send a memory address to the EEPROM chip and receive the contents of that address back. Receiving data is as easy as sending; it just operates in reverse:

```
//Clear Interrupt, Enable TWI, send ACK
TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWEA);
I2C_waitComplete();  //Wait for Transmission to
complete
Data = TWDR;         //Read data
```

Remember that you can only read data if the R/W bit of the address byte is set to a 1.

By now, you're probably getting familiar with clearing the interrupt flag and enabling TWI to start transmission. The ACK (**TWEA**) bit, however, is new. By setting this, we are telling the microcontroller to confirm back to the I²C module that we've received the data — remember, ACK is short for acknowledge. Surely we'd always want to let the I²C module know that we've received the data? Usually we do, but the ACK means something more. In addition to meaning it received the data, it can also mean "ready to receive more data." In other words, to terminate a transmission, you do NOT send an ACK.

Let's say you want to read a byte from a single memory location on the EEPROM; you will perform a read without an ACK. If, however, you want to sequentially read the data from the EEPROM, you will send an ACK after all but the last byte.

If this sounds confusing, it's because it is a little! Wait until you see it in action in the main code.

Send a Stop Condition

A "Stop Condition" ends the conversation — like putting the phone receiver down does. It also works with the Control register, and as you guessed, we need to:

```
TWCR = (1<<TWINT)|(1<<TWSTO)|(1<<TWEN);
```

Working with the EEPROM

The 24LC128 is a nice easy module to get started with — all it does is store and retrieve data. There are no settings or other behaviors to worry about. Let's see how typical transmissions would work.

Writing to EEPROM

Refer to **Listing 1** again. As expected, we kick-off with a Start condition. Next, we combine the address of the module with a bit indicating we want to perform a write, then send that down the wire. The module with that address prepares itself to receive data.

We then send a series of bytes over the line. The memory locations on the EEPROM are 16-bit locations, so we need to separate them into high and low bytes — shifting them just as we did in the June article. We send the memory location's high byte, then the low byte, and finally, the actual data we want to store at that location.

Our transmission is complete, so we issue a Stop condition.

Reading from EEPROM

If you're comfortable with writing to the EEPROM, then **Listing 2** won't frighten you — it shows how to read

```
uint8_t EEPROM_read(uint16_t deviceAddress,
uint16_t memoryAddress)
{

    uint8_t readData = 0;
    uint8_t result;


    //Send START Condition
    result = I2C_sendStart();

    //Send the device Address with WRITE - we
    //need to WRITE to specify the address to
    //read from
    result = I2C_send(deviceAddress|TW_WRITE);

    //Send Memory Location Address to read from
    //(High)
    result = I2C_send(memoryAddress >> 8);

    //Send Memory Location Address to read from
    //(Low)
    result = I2C_send((uint8_t)memoryAddress);


    //Send REPEATED START Condition - now we
    //read from the memory address
    result = I2C_sendStart();


    //Send the device Address with READ
    result = I2C_send(deviceAddress|TW_READ);

    //Read the data returned by the EEPROM
    readData = I2C_read(0);    //Read Data with
    NO ACK  (No ACK means we're done reading)

    //Send STOP Condition
    I2C_sendStop();

    return readData;
}
```

Listing 2. Function to read a byte from EEPROM.

		Tens			Units			
BIT	7	6	5	4	3	2	1	0
DESC	-	MINTEN2	MINTEN1	MINTEN0	MINONE3	MINONE2	MINONE1	MINONE0
05 mins:		0	0	0	0	1	0	1
15 mins:		0	0	1	0	1	0	1
57 mins:		1	0	1	0	1	1	1

Figure 8: Binary Coded Decimal – an example.

from EEPROM. We kick the read off in a similar way: Start condition, address with write bit set, send memory location. Wait – did that seem counter-intuitive? We set the *write* bit, but we want to *read* data!? The write and read bit don't refer to the operation we're performing at a module level (i.e., writing to memory or reading from memory), it refers to the operation being performed between the module and MCU. In this case, we need to *write* to the module in order to tell it which address we want to *read* from. Carry on reading through the code and it should make sense.

So we send the memory location (high and low bytes), and then we issue *another* start condition (called a repeated start). This tells the module we're going to do something different in the same conversation. Then, we re-address the module with the read bit set. This tells the module to get ready to send us the data we requested in our initial write. We read this data and once we've got what we want, we terminate the conversation with a Stop condition.

EEPROM is Mastered

You've now mastered the EEPROM – and got to grips with understanding how to use read and write. Remember that with most modules, you'll need to write to a device to tell it what you want to read. All the EEPROM-related code in the project is stored in the files "EEPROM.c" and "EEPROM.h" at the article link.

Your Turn – The RTC

Now that you've got a feel for how I²C works, I'm going to give you a turn to interpret the code for the RTC – you'll find it in "RTC_MCP79400.c" and "RTC_MCP 79400.h" also at the article link. Before I do that, though, there are a couple of principles I want to run through.

Registers

The RTC uses registers – and lots of them! There are registers to set seconds, registers to set and read minutes, hours, days, weeks, months, years, leap years, days of the week ... and then alarm registers, configuration registers ... you get the picture. The only way to master this module is to read the datasheet ... a number of times!

Register Names: As this is an external module, Atmel Studio doesn't know the register names. This means you'll need to define these yourself if you want to make your code readable. Take a look at the file "RTC_MCP79400.h" – it contains the address names and bit numbers for the most common registers ... and yes, I had to type them all in manually off the datasheet!

Register Values: To set a value in a register takes a couple of writes – one to specify the register you want to write to, and another to specify the value that goes into the register. To read from a register, you'll follow the same process as you did when reading from the EEPROM – except that the register address is only one byte, so it doesn't need to be split into high and low.

Changing Register Values: This is an important point. You can't change single bits in the RTC's registers like you would in your microcontroller's registers – if you write a single bit over I²C to a register, you clear all other bits in the register and only set that one bit. Not ideal! If you want to preserve the other bit values in a register, you'll need to read the register value, perform your bitwise logic on the value you've read, and then write the entire register back again.

Binary-Coded Decimal

Binary is binary, right? Why is the datasheet littered with references to "Binary-Coded Decimal?" Binary-Coded Decimal (BCD) is not as common as it was historically, but nevertheless is fairly common in RTCs. It is a way of using binary to represent individual digits in a number – usually four bits to represent a single digit (0-9). **Figure 8** shows a representation of the *RTCMIN* register (which stores the "minutes" portion of the time), and a few BCD examples.

To make things simpler (well, only slightly), there are a couple of helper functions in "RTC_MCP79400.c" that convert between BCD and decimal. Unfortunately, BCD is a reality, and I grumblingly learned to live with it. If BCD sounds like something that interests you, there is a pretty comprehensive article on Wikipedia that you can sink your teeth into.

Configuration Registers

There are a couple of important configuration registers that you need to know about, apart from the main timekeeping and alarm registers:

RTCSEC: Bit 7 of RTCSEC is "ST" – this bit starts the oscillator. Without an oscillator, your clock isn't ticking, so ensure that you set it on startup.

RTCWKDAY: This has a number of useful bits:

- **OSCRUN** tells you whether the oscillator is running (useful for fault checking and troubleshooting).
- **VBATEN** needs to be set to 1 if you're using a backup battery. Without this, the RTC doesn't know to

Parts List

Projects here are based on the build from "Beyond Arduino #1" in the March edition of *Nuts & Volts*. The following additional components are needed for the Temperature Logger project:

U1	LM35 Temperature Sensor
U2	Microchip 24LC128 EEPROM
U3	Microchip MCP79400 RTC
R1, R2, R3	10K Resistor
R4, R5	2.2K Resistor
Y1	32.768 kHz Tuning Fork Crystal
C1, C2	8.2 pF Ceramic Capacitor
C3	100 nF Ceramic Capacitor

A complete kit to go with this article can be purchased online from the *Nuts & Volts* Webstore @ www.nutsvolts.com or call our order desk at 800-783-4624.

switch to backup power when the main power fails.

Setting and Reading the Time

Time keeps marching on — which makes it difficult to read or set the time over an I²C interface. You need to find a way to "freeze" time in order to read it or set it. If you don't freeze time, you risk time "rolling over." Let's say you start reading the *time* at 11:59:59 pm on 15 June, and finish reading the *date* at 12:00 pm 16 June — an operation that could take milliseconds. Your resultant time will look like 11:59 pm on 16 June — a whole day out!

When reading the time, the RTC notices you're accessing the registers and copies the instantaneous time into a buffer. To make sure that you read a "frozen" copy of time from this buffer, you must read all the time elements you need in one operation (i.e., before you issue a Stop condition).

The same risk exists when setting the time. When setting the time, you must first halt the oscillator (setting ST on RTCSEC to "0"). Now that the clock has stopped, you can set the time before starting the oscillator ticking again (setting ST on RTCSEC to 1). Always check it has started up again (read the RTCWKDAY register and check the OSCRUN bit).

A Lot to Take In

We've covered a lot of ground in this article, and at quite a pace — not only the basics of I²C, but also the workings (and quirks) of working with EEPROM and RTC modules. I hope it has been useful. We've developed a fairly substantial little project that could be expanded in any number of ways. Thanks for joining me again, keep the feedback coming, and I look forward to some more exploration next time. **NV**

Behind the Scenes

I have to confess that projects of this size test my breadboarding patience! Too many jumper wires crossing, counting the MCU pins incorrectly too often, troubleshooting bad connections, finding jumpers mysteriously pulled out, and generally spending more time guarding my project than developing it!

Figures A and **B** show how I worked on this project to get it running quickly — a series of boards I developed to make my life easier (and, of course, fun). The entire temperature logger is contained in the three PCBs and the few components on the breadboard. I was chatting with the team at *Nuts & Volts*, and we thought that these boards could also make reader's lives easier. Keep an eye on the *Nuts & Volts* store where we'll put a few up to see if there's interest. The main board is called a "Toadstool" and the two plug-in modules are called "Caps" — the Toadstool contains the ATmega328P, and the two removable caps you see here contain the MCP79400 and the 24LC128.

Let us know your thoughts!

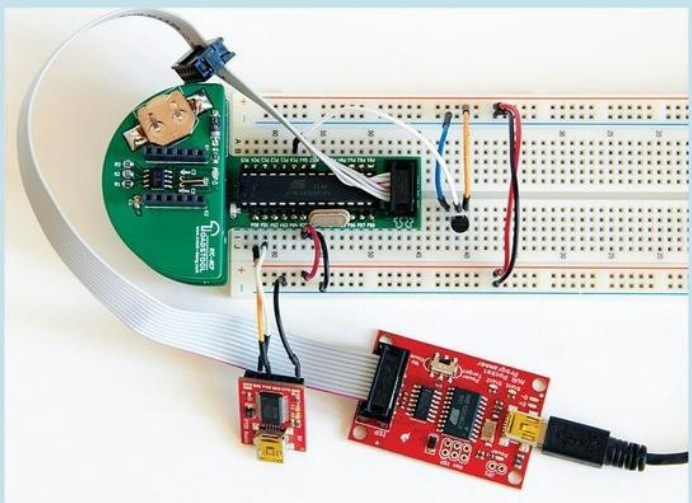


Figure A: Temperature Logger using a Toadstool.

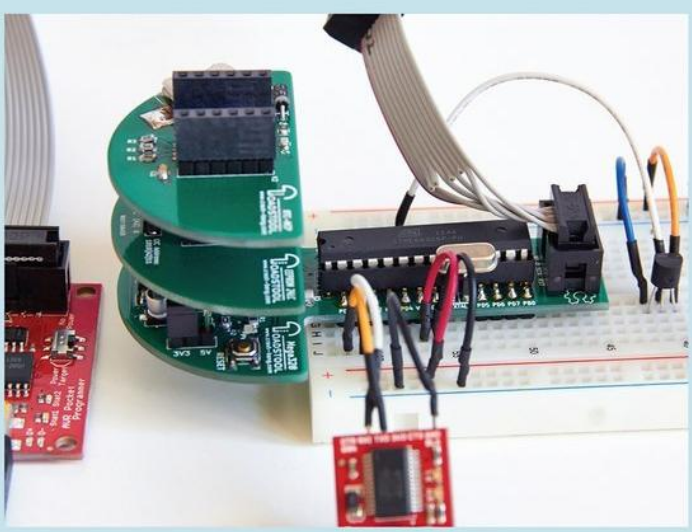


Figure B: Stacking the caps: EEPROM and RTC on the Toadstool.

■ BY WARD SILVER N0AX
hwardsil@gmail.com

RF Interference

It's everywhere! It's everywhere! Fortunately, you can take a bite out of RFI.

RF interference — is it interference to you? Is it interference by you? Possibly both! What does this interference consist of? And how can you tell what type is present? A topic that starts off with so many questions is bound to cover a lot of ground, so let's get started.

The phenomenon known as RF interference — RFI to its many friends and acquaintances — is a sub-type of electromagnetic interference, or EMI, which is itself a topic that is part of the general subject of electromagnetic compatibility, or EMC. Specifically, RFI involves interference caused by signals propagated wirelessly as radio waves, but even this sub-type of a sub-type of a subject still has a surprisingly subtle and broad reach.

Is RFI Really a Problem?

Undoubtedly. It's a growing problem beginning to have an impact on the broadcasting and wireless data/telephony industries. Electromagnetic "smog" is pervasive in urban and suburban areas across the entire frequency range, from long-wave signals through microwaves.

The most widespread effect is higher random noise levels all across the spectrum, just as atmospheric smog affects an entire region. This raises the noise floor (the noise present at any frequency), meaning weak signals are increasingly obscured and stronger signals degraded by bits and pieces.

Why would, say, a mobile telephony company care about analog noise since the phones communicate digitally? Because, at some level, just about everything is analog. Inside that phone is a radio receiver looking for analog radio signals from the local base station.

If I mix in a little more noise — let's say, I raise the noise level by a measly 3 dB at the receiver input — the distance at which the phone's digital protocol breaks down will decrease by about 30%. That means the coverage area for the base station has decreased by half!

The service provider then has to either build more towers or you have to transmit at a higher power, or both. That's money out of your pocket, one way or the other. The same is true for stations broadcasting programming, or public safety agency communications.

On a personal basis, most people don't know (or care) about RFI until their garage door opener won't work or an audio system starts speaking in tongues. As readers of *Nuts & Volts*, however, you should become aware of the problem, understand its causes, and know how to solve it. As day-to-day users of the spectrum, ham radio folks are intimately familiar with the symptoms and cures for RFI.

RFI Basics

The definition of RF itself (radio frequency) spans electromagnetic signals over a range of just above audio (about 20 kHz) through infrared light at several hundred THz. To keep this column from occupying the entire magazine, we'll stay in the range of 500 kHz through the upper end of UHF (ultra-high frequency) around 3 GHz — still a 2,000-fold range.

RFI is discussed as having sources and victims with a path between them. The relationship should be pretty obvious. The goal — paraphrasing *Poor Richard's Almanac* — is to "neither a source or victim be." To deal with RFI effectively, you must be able to identify all three:

1. What system or device is creating the interfering signals and how — the source?
2. What system or device is being interfered with and how — the victim?
3. What is the path by which the RF is radiating from the source and getting to the victim?

Even across the many octaves, the basic techniques to achieve the goal are the same: Keep RF out and keep RF in. The techniques for implementing those two seemingly simple tactics are themselves deceptively simple sounding: Avoid and block.

What is the Pin 1 Problem?

In the pro audio world, the "pin 1 problem" was identified by Neil Muncie in 1994. It is caused by the shield pin of the ubiquitous XLR connector (pin 1) being attached directly to circuit common. This connection provides an RF highway for signals being picked up on the cable's shield directly into the equipment where they can cause interference. Simultaneously, any noise or spurious signals present on the circuit common are given a terrific antenna to radiate from.

The Mackie Company — a professional audio equipment manufacturer — has addressed this in the application note, "Grounds, Shields, Hums, and Buzzes" at **www.mackie.com/pdf/CMRefGuide/Tips_Ch6.pdf**. More information is available in several papers and tutorials by Jim Brown K9YC (a retired professional audio engineer) on his website at **http://audiosystemsgroup.com/publish.htm**.

Differential and Common-Mode Signals

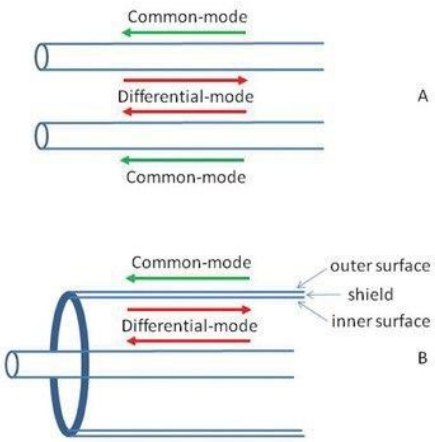


FIGURE 1. Common-mode (CM) and differential-mode (DM) signals can flow on parallel conductor and coaxial cables at the same time without mixing. None of the conductors need to be grounded for RF to pick up and/or radiate RF common-mode signals.

Before digging too deep into cause and effect, we have to understand the two types of signals we'll be dealing with: differential-mode (DM) and common-mode (CM). **Figure 1** shows DM and CM signals for typical unshielded paired wires at A, and a coaxial cable at B. Remedies for DM signals probably won't work for CM and vice versa.

DM signals are what we often think of as "balanced" in that the signal consists of identical currents flowing in opposite directions along two closely-matched paths — neither of which is connected to a ground or grounded enclosure. CM signals flow equally on all conductors of a multi-conductor cable or on the common (shield) conductor of a coaxial or shielded cable.

Taking a close look at **Figure 1B**, you can see that a single cable can support both DM and CM signals *at the same time*. In fact, at RF, the outside of a braided or foil shield is electrically independent from the inside! That's why we use shielded cables — to keep noise and radiated signals from being picked up by the conductor inside the shield.

Why does this happen? The *skin effect* restricts AC current flow to the outside of a conductor. As shown in **Figure 2**, at frequencies above 1 MHz, current penetrates copper or aluminum less than 0.1 mm!

What Causes RFI?

Interference itself can take a variety of forms. If you are a radio user — as hams are — interference may just be higher noise levels or it can be an actual spurious signal or spur that obscures a desired signal or upsets receiver operation. For example, the *harmonic* of an intentionally generated signal at an integer multiple of a *fundamental* frequency can fall on the same frequency as a desired signal thereby interfering with reception.

Spurious signals are a common problem when RF leaks out of equipment due to improper shielding or cabling. Digital signals that transition very quickly between voltage levels are composed of fundamental and many harmonics to generate the sharp edges. The harmonics appear all across the RF spectrum into the microwave region for high speed data. Once radiated, they propagate just like any other intentionally radiated signal.

Switchmode or switching power supplies or power converters are another very common source of spurious signals. These power converters work by "charging up" inductors with magnetic energy, then suddenly interrupting the current to transfer the energy to an output filter capacitor where it is changed to steady DC.

The interruption of current creates a wide spectrum of spurious signals spaced at intervals of the supply's switching frequency. If output filtering is not designed properly or — in a growing number of instances for imported supplies — not installed at all, the strong spurious signals can disrupt normal communication throughout an entire neighborhood.

Common uses of switching converters include lightweight wall wart DC supplies, battery chargers, electronic lighting ballasts, and low voltage lighting.

Really strong signals can overload a receiver to the point where it causes distortion of a desired signal — even if the signal is completely legal and not in the frequency range being received. This is called fundamental overload and is caused by the receiver's inability to reject the out-of-band signal. Inexpensive receivers such as wireless phones or portable radios have limited filtering, so are susceptible to this type of interference.

Maybe a receiver isn't involved at all and the interfering signal is simply so strong that it disrupts the proper operation of an electronic

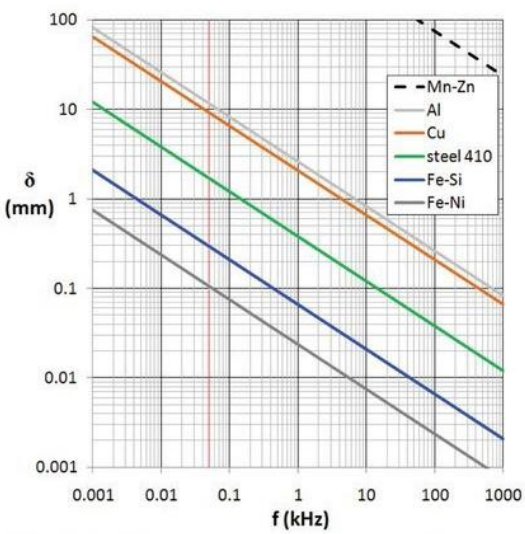


FIGURE 2. Skin depth vs. frequency for some materials. Red vertical line denotes 50 Hz frequency: Mn-Zn, magnetically soft ferrite; Al, metallic aluminum; Cu, metallic copper; steel 410, magnetic stainless steel; Fe-Si, grain-oriented electrical steel; Fe-Ni, high-permeability permalloy (80%Ni-20%Fe).

Cable System Leakage

Cable TV systems carry signals from the very low MHz to upper UHF, so extra attention has to be paid to the details of making proper connections. This keeps the cable system's programs noise-free, and also prevents the signals getting out and causing interference. The cable TV channels are not restricted in frequency like over-the-air TV, so the signals can be directly on top of other over-the-air signals. This is known as cable leakage and is most often due to loose, dirty, or improperly installed type F connectors on the cable feed. If you experience interference to or from your cable TV equipment, making sure connectors are tight, clean, and properly installed is a great place to start looking for problems.

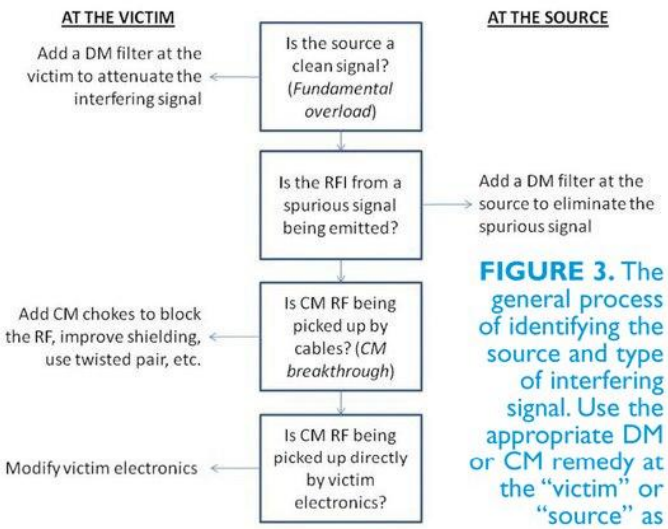


FIGURE 3. The general process of identifying the source and type of interfering signal. Use the appropriate DM or CM remedy at the "victim" or "source" as indicated.

device. We can refer to this as common-mode breakthrough, where the interfering signal is picked up by cables and carried to internal electronics where it is strong enough to alter circuit functions.

It is also common for unshielded electronics in plastic boxes to pick up and be disrupted by strong signals without any cables at all! This is direct detection and it can require modifications to the circuit to eliminate the interference.

The general process for assessing which types of RFI one is dealing with is outlined in the steps of **Figure 3**. There are different remedial techniques for each. Using the wrong technique or applying it to the wrong end of the path won't solve the problem.

Keeping RF In

At a potential source — such as digital computer and networking equipment, microprocessor-controlled appliances, switching converters, any circuits that turn large currents on and off — the first step is to avoid radiating RF if you don't have to. This is not just being altruistic, because if your system is radiating RF it can also *receive* RF by the very same path. Closing off that path to outbound RF usually closes it to incoming RF, too! It's just good engineering to do that right in the design.

A good strategy to follow is to be sure your printed circuit board (PCB) has filtering applied to any signal or power connection entering or leaving the board. For example, a 0.01 µF disc ceramic capacitor to circuit common on a low frequency signal or control lines and power connections will act like a short-circuit at RF. That path will be "cold" at RF for both incoming and outgoing RF signals. (Note that any component connected directly to the AC line must be "line rated" to avoid fire hazards.)

Treat your equipment as if all of the signals going in and out of it were water, and every connection had to be water-tight. Start with a metal enclosure — even for simple projects. Using a plastic box means you have two strikes against you RF-wise: You have no shield that will route noise and RF currents away from the circuit, and every wire and PCB trace in the electronics will act as an antenna.

Use shielded cables and connectors — including a shielded AC power connector — with the shield connected to the enclosure.

Another often overlooked strategy is to avoid connecting the circuit common to any conductor that will leave the enclosure without being inside a shield. Keep the circuit common connected to the shielding enclosure as a separate connection from any data or signal cable as shown in **Figure 4**. (Refer to the sidebar, "What is the Pin 1 Problem?")

Keeping RF Out

Let's say the RFI is being caused by a receiver — such as an over-the-air TV receiver or scanner — by fundamental overload from an otherwise legitimate signal.

For example, you might live across the street from the local TV and FM antenna tower — somebody has to! The strong signal is probably coming right in the antenna input of the receiver, along with the signals you want to hear.

In this case, you can block the unwanted signal with a DM filter in the antenna input cable. The goal is to reduce the level of the signal you don't want until it is weak enough for the receiver to reject.

To do this successfully, you'll need to know the frequency of the signals you want and of the signal you don't want. If the unwanted signal is at a higher frequency, you can use a *low-pass filter* to allow the signals you want into the receiver while blocking the higher frequency unwanted signal. A *high-pass filter* does the opposite.

In some circumstances, you might have to use a *band-pass filter* that only allows some frequencies through while blocking all others. Remember that if the unwanted signal is on the same frequency as

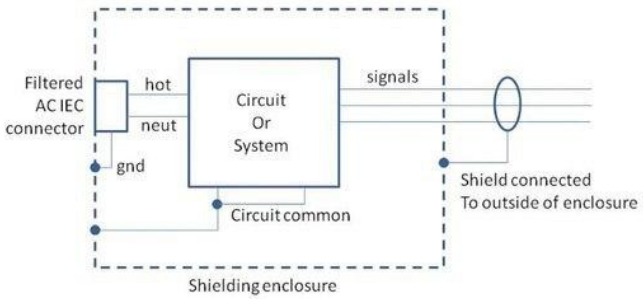


FIGURE 4. For RF immunity and to avoid radiated RF, electronics should be constructed inside a conductive enclosure such as aluminum. Connectors should be shielded, with the shield connected to the outside of the enclosure.

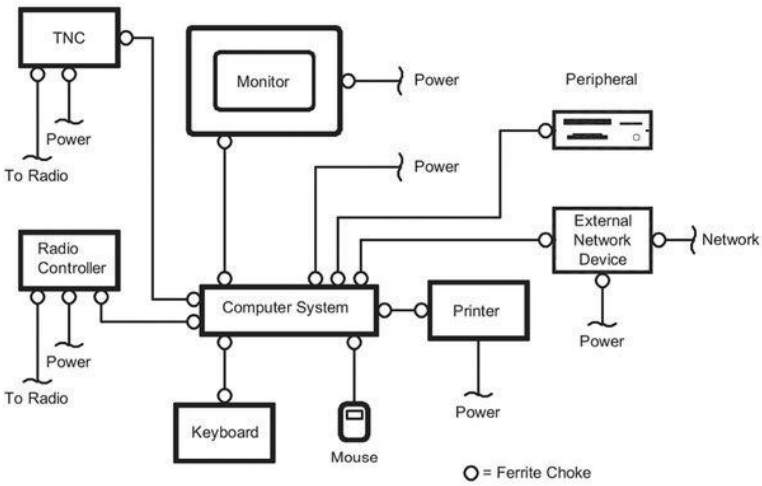


FIGURE 5. The use of ferrite chokes in a computer system where any piece of equipment can either be an RFI source or RFI victim. Ferrite chokes should be placed as close to the equipment as possible. *Graphic courtesy of ARRL.*

the one you do want, no filtering is possible. You'll have to figure out how to get rid of the interfering signal at the source.

In the previous column, I talked about ferrite cores and beads, and how great they were for suppressing RFI. Ferrite chokes — placed at the right location in a system of electronics — can do wonders at blocking CM signals getting into your system on the shields and common-mode paths of wiring and cables.

Figure 5 shows a typical computer in which any piece of equipment can be generating and receiving noise — a distressingly common situation — with recommended locations for the chokes.

Ferrite chokes consist of a clamp-on core or a toroid with cables or wires wound on them. Don't be satisfied with just passing a wire through a core once. Wind as many turns on the core as you can to increase the impedance created by the choke.

Remember from the previous column that ferrite impedance changes with frequency, so be sure to use the right type or mix of ferrite. If you are mostly concerned with RF below 30 MHz, use a type 31 mix ferrite.

For higher frequency signals, use type 43. (The various recommendations are covered in the Fair-Rite catalog, available online at **www.fair-rite.com**.)

More About RFI

I've just barely scratched the surface of a topic to which engineers devote entire careers! You may just want to get rid of some interference or keep a system from interfering with some other system. Or, maybe you are designing for reliable operation — good for you!

Hams have been dealing with RFI since the first transmitter was built, so you can find a great deal of useful references on the ARRL's RFI website for hams at **www.arrl.org/radio-frequency-interference-rfi**. Jim Brown's tutorials mentioned in the sidebar are good, and there is an excellent chapter on RF interference in the *ARRL Handbook's* recent editions.

Design for EMI quality — to keep the RF both in and out — at the outset of building your system. You won't regret it! **NV**


6th
ANNUAL
WORLD
NEW YORK
HALL OF SCIENCE
Maker Faire®
GREATEST
SHOW & TELL
ON EARTH
SEPT 26+27
SAT & SUN 10AM–6PM
NEW YORK
HALL OF SCIENCE
QUEENS, NY
GET YOUR
TICKETS
TODAY!
makerfaire.com
BROUGHT TO YOU BY
Make: MAGAZINE
GOLDSMITH SPONSORS
Disney
Google
intel
ITA
ITALIAN TRADE AGENCY
Microsoft
SILVERSMITH SPONSORS
Atmel
Cognizant
DREMEL
Makeblock
NASA
sprout by hp
MEDIA SPONSOR
NUTS VOLTS

ELECTRONET

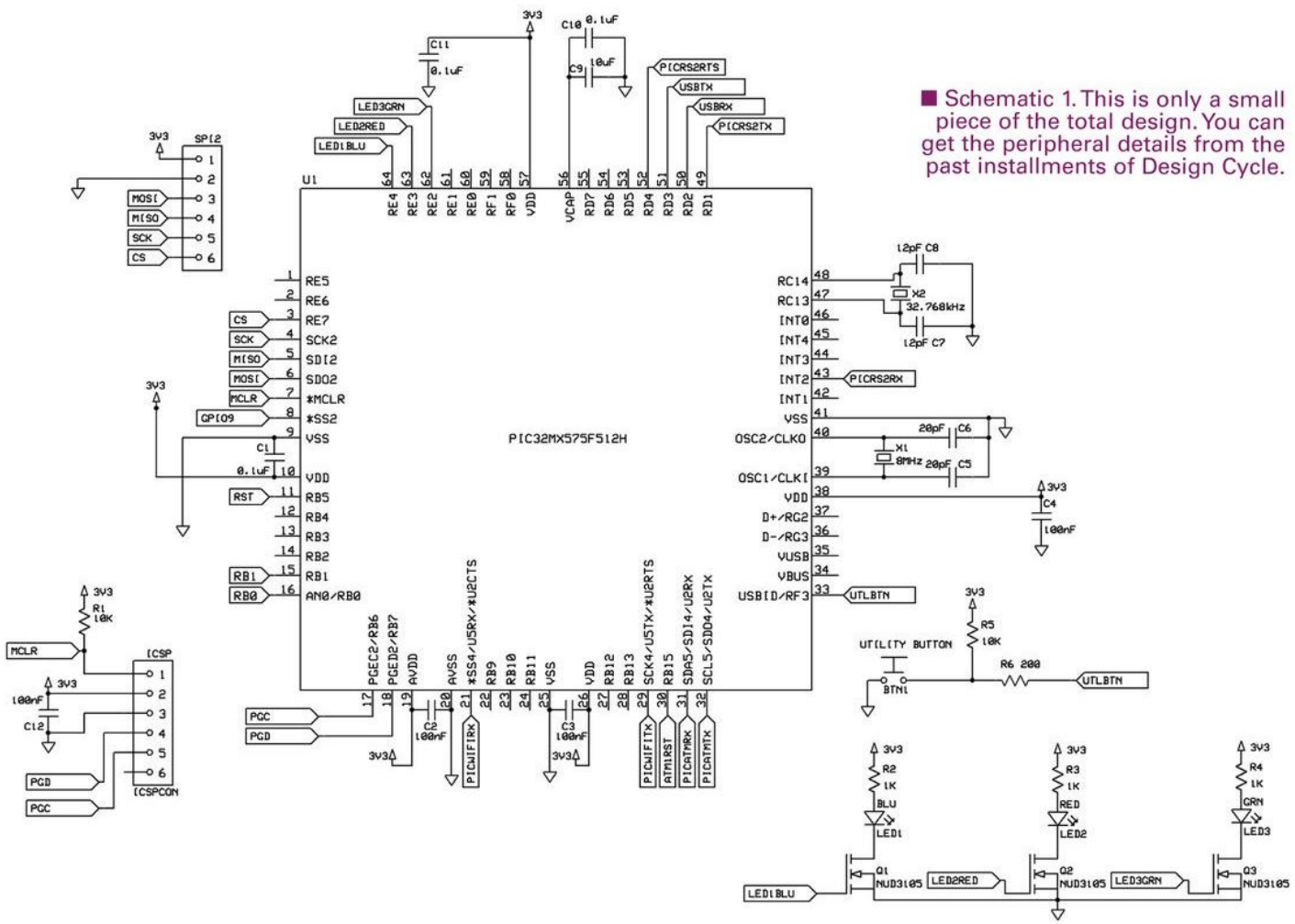
■ BY FRED EADY

Take Your PIC of a Super-Fast Embedded Computing Machine

Here lately, I've been in a PIC32MX design mode. In the past two installments of Design Cycle, I've presented a couple of unique PIC32MX hardware designs. We covered the hardware aspects of both designs with a promise of returning to do the same on the firmware side. We'll recap both PIC32MX hardware designs and then dive into writing the firmware to drive the PIC32MX peripherals.

The PIC32MX575F512H Design

Schematic 1 is a graphical depiction of the microcontroller component of the PIC32MX575 hardware design. Beginning at pin 1 and moving counter-clockwise around the PIC32MX595F512H, we encounter our first potential firmware candidate which happens to be an SPI portal. As you can see in **Photo 1**, we don't have an



■ Schematic 1. This is only a small piece of the total design. You can get the peripheral details from the past installments of Design Cycle.

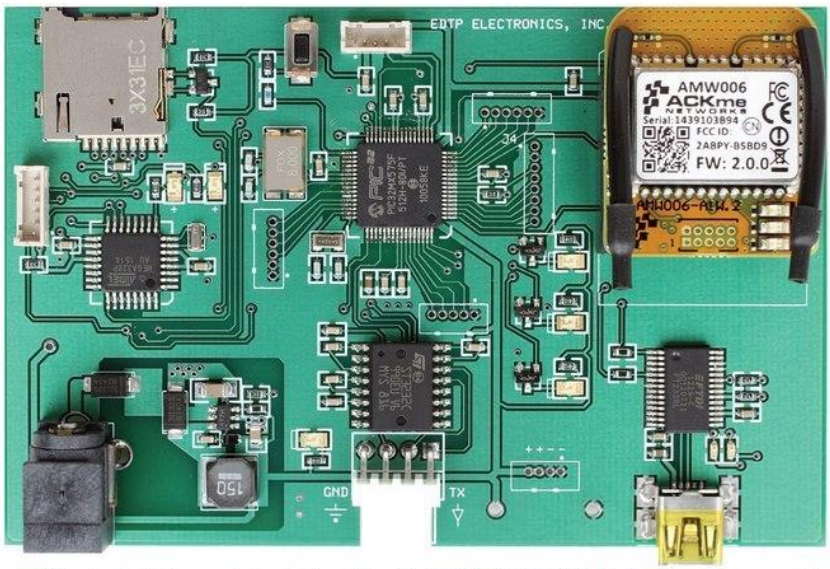
external SPI peripheral to drive at this time. All we can really do right now is activate the SPI portal pins. So, we'll forego the SPI driver.

The next peripheral we come upon is a serially interfaced Wi-Fi module. We just happen to have an ACKme Hopper sitting in the dual 10-pin socket space. We can also drive an XBee module or a Microchip WiFly module from the same socket space. Naturally, each radio variant would require its own unique firmware driver.

This design incorporates an instance of the OpenLog. All of the microSD card heavy lifting is done by the open source code that is loaded in the Atmel microcontroller. The OpenLog was originally designed to be used as a "no brainer" data logger. Thanks to a simple read-write command set — which is accessible via a standard three-wire serial connection — we can steer the OpenLog as if it were a filesystem.

Like most all of my PIC32MX designs, this particular piece of 32-bit hardware includes a 32.768 kHz crystal that feeds the PIC32MX575F512H's on-chip RTCC (Real Time Clock Calender). In applications that do not require a clock/calendar function, the PIC32MX575F512H's RTCC peripheral can be used to alarm and interrupt at predetermined intervals.

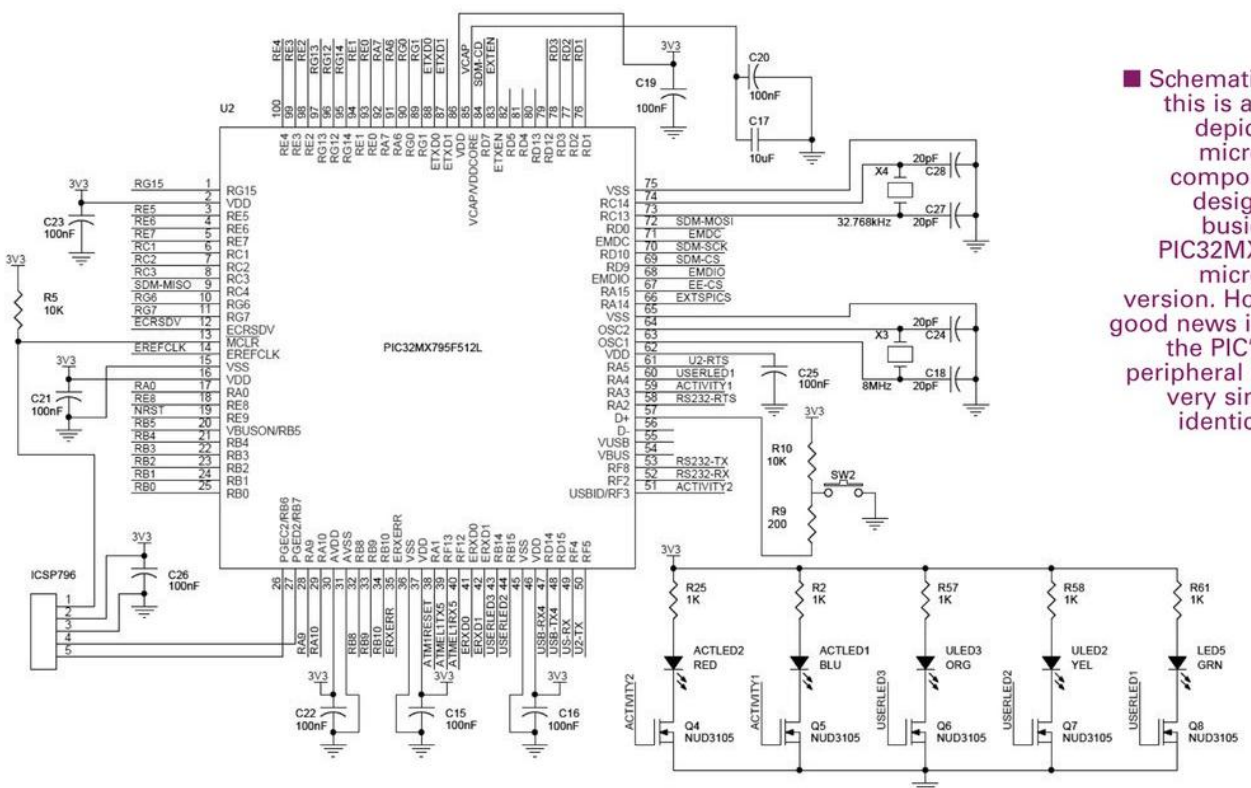
Moving our attention back to **Photo 1**, we can conclude that we will be writing drivers for four serial ports. Along the way, we'll most likely need to blink some LEDs and read a pushbutton.



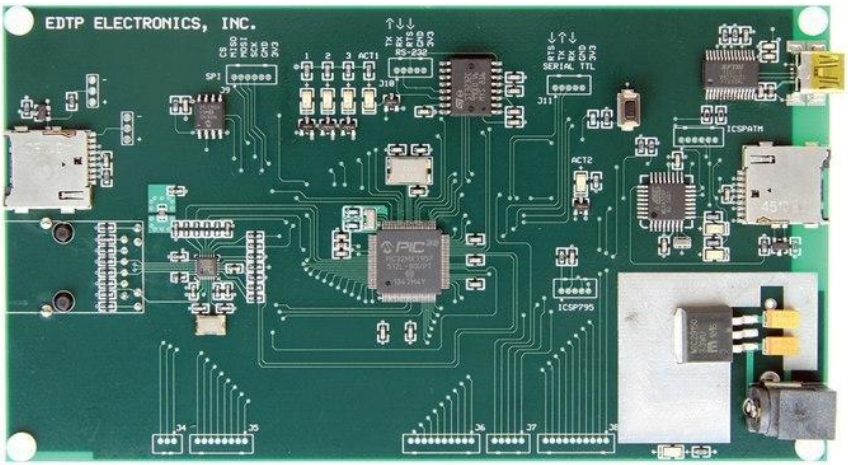
■ Photo 1. This aerial shot of the PIC32MX575F512H hardware reveals the need for microSD, RS-232, and Wi-Fi module driver firmware.

The PIC32MX795F512L Design

Schematic 2 is a bit busier than its MX575F512H cousin. A quick look at **Photo 2** tells us that there are peripherals which are common to both designs. The FTDI USB port, the OpenLog circuitry, and the RS-232 port are



■ Schematic 2. Again, this is a schematic depiction of the microcontroller component of the design. It's a bit busier than the PIC32MX575F512H microcontroller version. However, the good news is that both the PIC's firmware peripheral drivers are very similar, if not identical in some cases.



■ Photo 2. This hardware design gains access to a LAN and the Internet via a wired Ethernet connection. An XBee or Wi-Fi module can be accommodated simultaneously at the SERIAL TTL interface.

identical to their PIC32MX575F512H counterparts. This design also includes a 32.768 kHz crystal to drive the PIC32MX795F512L's RTCC module.

The major difference between our PIC32MX575F512H and PIC32MX795F512L designs is the Ethernet hardware implementation. We don't have to scratch out an Ethernet driver as the folks at Microchip have already done that for us. That takes care of the 25LC1024 EEPROM and the microSD card that is parked next to the Ethernet magnetics.

I've tested the Ethernet hardware successfully using the Ethernet demo applications found in the Microchip MLA v2013-06-15. MLA is short for Microchip Libraries for Applications. If you're into Microchip's Harmony, you should also be successful in establishing an Ethernet session with this hardware design.

USART Firmware

Let's begin by setting up the USARTs. We only have to write this code once as the USART setup procedure is identical for both PICs. Our USART drivers will transmit on demand and receive via interrupt. The receive interrupt mechanism is assisted by a ring buffer. Each USART we activate will be attached to its own ring buffer. A ring buffer head pointer and tail pointer byte will also be assigned:

```
//************************************
//* USART RECEIVE BUFFERS SETUP
//* 1,2,4,8,16,32,64,128 or 256 bytes
//************************************
#define USART1_RX_BUFFER_SIZE  256
#define USART1_RX_BUFFER_MASK (
USART1_RX_BUFFER_SIZE - 1 )
BYTE USART1_RxBuf[USART1_RX_BUFFER_SIZE];
BYTE USART1_RxTail;
BYTE USART1_RxHead;
```

Each USART ring buffer will consist of 256 bytes. We will initialize four 256-byte ring buffers. That's just a drop in the proverbial RAM bucket as the PIC32MX575F512H is endowed with 64 KB of on-chip RAM. The remaining USARTs are defined identically to USART1. All we change is the USART number. For instance, here's how we set up USART5:

```
//************************************************
//* USART RECEIVE BUFFERS SETUP
//* 1,2,4,8,16,32,64,128 or 256 bytes
//************************************************
#define USART5_RX_BUFFER_SIZE  256
#define USART5_RX_BUFFER_MASK (
USART5_RX_BUFFER_SIZE - 1 )
```

```
BYTE USART5_RxBuf[USART5_RX_BUFFER_SIZE];
BYTE USART5_RxTail;
BYTE USART5_RxHead;
```

The FTDI USB-to-serial IC utilizes USART1. USART2 is dedicated to the Atmel ATmega328P that handles the microSD drive. USART4 is electrically connected to the ST3232 RS-232 driver/receiver. The ACKme Wi-Fi module communicates with the PIC32MX575F512H on the USART5 interface.

USART Setups

Once the USART data memory is allocated, we can power up each USART module and define their associated interrupt mechanisms. The USARTs all have some common settings such as data length, parity, and the number of stop bits. However, enabling the individual interrupt engines is slightly different for each USART. As you would imagine, the USART initialization takes place in the hardware initialization function. Here's how we set up USART1:

```
//Initialize UART1  FTDI
U1BRG = 43; // Set Baud rate 115200 bps
U1STA = 0;
U1MODE = 0x00008000;
   // Enable UART for 8-none-1
U1STASET = 0x00001400;
   // Enable Transmit and Receive
//priority 2 sub priority 3 - 00011111 bits
//<4:2> sub bits <0:1>
IPC6SET = 0x0000000B;
IEC0SET = 0x08000000;
IFS0CLR = 0x08000000;
// flush receive buffer
USART1_RxTail = 0x00;
USART1_RxHead = 0x00;
```

The interrupt priority and sub-priority bit masks can be found in the interrupts section of the PIC32MX575F512H datasheet. If you like your turnip greens from a can, you can also get these definitions in a more descriptive form from the PIC32MX575F512H *include* file that is part of the Microchip XC32 C compiler. The rest of the USARTs are initialized in this manner:

```
//Initialize UART2 ATMega328P - microSD
U2BRG = 520; // Set Baud rate 9600
U2STA = 0;
U2MODE = 0x00008000;
   // Enable UART for 8-none-1
U2STASET = 0x00001400;
   // Enable Transmit and Receive
//priority 2 sub priority 3 - 00011111 bits
//<4:2> sub bits <0:1>
IPC8SET = 0x0000000B;
IEC1SET = 0x00000200;
IFS1CLR = 0x00000200;
// flush receive buffer
USART2_RxTail = 0x00;
USART2_RxHead = 0x00;

//Initialize UART4 RS-232
U4BRG = 520; // Set Baud rate 9600
U4STA = 0;
U4MODE = 0x00008000;
   // Enable UART for 8-none-1
U4STASET = 0x00001400;
   // Enable Transmit and Receive
//priority 2 sub priority 3 - 00011111 bits
//<12:10> sub bits <9:8>
IPC12SET = 0x00000B00;
IEC2SET = 0x00000010;
IFS2CLR = 0x00000010;
//flush receive buffer
USART4_RxTail = 0x00;
USART4_RxHead = 0x00;

//Initialize UART5 WiFi
U5BRG = 43; // Set Baud rate 115200
U5STA = 0;
U5MODESET = 0x00008000;
   // disable UART for 8-none-1
```

```
    U5STASET = 0x00001400;
       // Enable Transmit and Receive
    //priority 2 sub priority 3 - 00011111 bits
    //<26:28> sub bits <25:24>
    IPC12SET = 0x0B000000;//0000 1011 0000 0000
    0000 0000 0000 0000
    IEC2SET = 0x00000400;
    IFS2CLR = 0x00000400;
    //flush receive buffer
    USART5_RxTail = 0x00;
    USART5_RxHead = 0x00;
```

A Bit of Redirection

The FTDI USB port will most likely be used to pass information between a PC and the PIC32MX575F512H. The ST3232 is present to make sure we can talk to legacy RS-232 equipped gadgets. If the hardware is used to communicate via a LAN or the Internet, we will be "speaking" through the ACKme Wi-Fi more often than not.

So, to make it a bit easier to pass data to the Wi-Fi module, we will redirect the *STDOUT* to USART5. This redirection allows us to use the *printf* statement to send data to the USART5 portal. It's really easy to do. Here's the redirection code:

```
//*********************************************
//* UART5 REDIRECTOR
//*********************************************
void _mon_putc(char c)
{
       U5TXREG = c;
       while(U5STAbits.TRMT == 0);
}
```

The redirection can be applied to any of our USARTs in a mutually exclusive manner by simply changing the USART number inside of the *mon_putc* function.

USART Interrupt Handlers

Putting the received data into the ring buffers is accomplished with our USART receive interrupt handlers. Each USART has one. Here's the USART5 interrupt handler:

```
//*********************************************
//* UART 5 interrupt handler
//* it is set at priority level 2 with software
//* context saving
//*********************************************
void __ISR(_UART_5_VECTOR, IPL2SOFT)
IntUart5Handler(void)
{
       BYTE data,tmphead,tmptail;

       data = U5RXREG;
       // read the received data
       // calculate buffer index
       tmphead = ( USART5_RxHead + 1 ) &
       USART5_RX_BUFFER_MASK;
       USART5_RxHead = tmphead;
       // store new index

       if ( tmphead == USART5_RxTail )
       {
       // ERROR! Receive buffer overflow
       // Your error handler code goes here
       }

       USART5_RxBuf[tmphead] = data;    // store
received data in buffer
       IFS2CLR = 0x00000400; //0000 0000 0000
0000 0000 0100 0000 0000
}
```

Each USART interrupt handler is identical with the exception of the *IFS2CLR* interrupt flag bit mask. You can see this in the USART4 interrupt handler code:

```
//*********************************************
//* UART 4 interrupt handler
//* it is set at priority level 2 with software
//* context saving
//*********************************************
void __ISR(_UART_4_VECTOR, IPL2SOFT)
IntUart4Handler(void)
{
       BYTE data,tmphead,tmptail;

       data = U4RXREG;
       // read the received data
       // calculate buffer index
       tmphead = ( USART4_RxHead + 1 ) &
       USART4_RX_BUFFER_MASK;
       USART4_RxHead = tmphead;
       // store new index

       if ( tmphead == USART4_RxTail )
       {
       // ERROR! Receive buffer overflow
       }

       USART4_RxBuf[tmphead] = data;
       // store received data in buffer
       IFS2CLR = 0x00000010;
}
```

USART Character Handlers

We must also have a way of retrieving the buffered data. That is done with the USART *recvchar* functions:

```
BYTE recvchar5(void)
{
  BYTE tmptail;
  // wait for incoming data
  while ( USART5_RxHead == USART5_RxTail );
  // calculate buffer index
  tmptail = ( USART5_RxTail + 1 ) &
  USART5_RX_BUFFER_MASK;
  USART5_RxTail = tmptail;
       // store new index

  return USART5_RxBuf[tmptail];
       // return data
}
```

Again, all of the USART *recvchar* functions are identical and delineated simply by the USART number:

```
BYTE recvchar1(void)
{
  BYTE tmptail;
  // wait for incomming data
  while ( USART1_RxHead == USART1_RxTail );
  // calculate buffer index
  tmptail = ( USART1_RxTail + 1 ) &
  USART1_RX_BUFFER_MASK;
  USART1_RxTail = tmptail;
```

```
        // store new index

    return USART1_RxBuf[tmptail];
        // return data
}
```

We check for characters in the ring buffers that need to be processed by calling the *CharInQueue* functions:

```
BYTE CharInQueue1(void)
{
  return(USART1_RxHead != USART1_RxTail);
}
BYTE CharInQueue2(void)
{
  return(USART2_RxHead != USART2_RxTail);
}
BYTE CharInQueue4(void)
{
  return(USART4_RxHead != USART4_RxTail);
}
BYTE CharInQueue5(void)
{
  return(USART5_RxHead != USART5_RxTail);
}
```

Precision Timing

At this point, we almost have everything we need to communicate using any one of the serial ports. It would be nice to be able to delay precisely and flash some LEDs if we have to. We will use the PIC32MX575F512H's core timer to build our millisecond delay function:

```
#define GetSystemClock()              80000000UL
#define GetPeripheralClock()          80000000UL
#define CoreTicksPerMs   GetSystemClock() / 2000)

//*********************************************
//* DELAY MILLISECOND FUNCTIONS
//*********************************************
void delayms(unsigned int ms)
{
       unsigned int msDelayTime, currentTickCnt;
       currentTickCnt = ReadCoreTimer();
       msDelayTime = (CoreTicksPerMs * ms) +
       currentTickCnt;
       while((ReadCoreTimer()) < msDelayTime);
}
```

Eye Candy

Blinking LEDs is my specialty. The PIC32MX575F512H's native SET, CLR, and INV atomic operations make manipulating the PIC32MX575F512H's I/O pins a walk in the park:

```
/** LEDS ***********************************/
#define bluLED
LATEbits.LATE4
#define bluLED_Off          LATECLR = 0x0010;
```

```
          //0000 0000 0001 0000
#define bluLED_On             LATESET = 0x0010;
          //0000 0000 0001 0000
#define bluLED_Toggle         LATEINV = 0x0010;
          //0000 0000 0001 0000

#define redLED                LATEbits.LATE3
#define redLED_Off            LATECLR = 0x0008;
          //0000 0000 0000 1000
#define redLED_On             LATESET = 0x0008;
          //0000 0000 0000 1000
#define redLED_Toggle         LATEINV = 0x0008;
          //0000 0000 0000 1000

#define grnLED                LATEbits.LATE2
#define grnLED_Off            LATECLR = 0x0004;
          //0000 0000 0000 0100
#define grnLED_On             LATESET = 0x0004;
          //0000 0000 0000 0100
#define grnLED_Toggle         LATEINV = 0x0004;
          //0000 0000 0000 0100
```

Checking Our Work

Now that we've written all of this code that supposedly allows us to rule the PIC32MX575F512H's gaggle of USARTs, let's try it out. Since the ACKme module is now available to us, let's write some code to wake it up, connect to a remote server, and send some data:

```
init();

//enter COMMAND mode from STREAM mode
grnLED_On;
delayms(300);
printf("$$$");
delayms(300);

//wait for Command Mode Start message -
//ends with >
indx5 = 0;
while(!CharInQueue5());
      do{
          rxBuf5[indx5++] = recvchar5();
        }while(CharInQueue5());
//issue tcp client command
if(rxBuf5[20] == '>')
{
      printf("tcp_client nemo.dyndns-
      server.com 8032\r\n");
      delayms(1000);
}
//exit to STREAM mode
printf("exit\r\n");
```

The ACKme Hopper has been preconfigured to boot into STREAM mode instead of COMMAND mode. Sending the trio of $ characters within the millisecond delays will force the Hopper to enter COMMAND mode and spit out a bunch of characters by doing so (the Command Mode Start Message). I've calculated that the command prompt (>) is always found at the 20th position of the receive ring buffer following the $$$ sequence. At this point, we can issue Hopper commands.

In this case, we want to contact a remote server device on the Internet which is identified as **nemo.dyndns-server.com**. The nemo server is listening behind port 8032. Instead of attempting to contact the nemo server — which is exclusive to EDTP Electronics — you can substitute an IP address and port number that is on your LAN. For instance, to reach a host on your LAN located at 192.168.0.222 listening on port 8088, code this:

```
if(rxBuf5[20] == '>')
{
    printf("tcp_client
    192.168.0.222 8088\r\n");
    delayms(1000);
}
```

The Hopper must be in STREAM mode to transfer data. So, a simple *printf* statement is all we need to get back.

Once back in STREAM mode, we can send data to the Hopper using *printf* or by stuffing the transmit buffer of USART5:

```
printf("Design Cycle\r\n");
//sends Design Cycle over WiFi
//followed by carriage
//return/line feed

{
   U5TXREG = recvchar1();
//sends characters over WiFi
//received at the FTDI USB
//port
   while(U5STAbits.TRMT == 0);
}
```

All of the incoming data from the Hopper is handled by the ring

buffer and interrupt mechanism we established for USART5.

A Good Day's Coding

Everything we have coded thus far can be reused by either the PIC32MX575F512H or PIC32MX795F512L. The download package at the article link that accompanies this discussion contains all of the source code we produced, plus all of the necessary code to bring up a PIC32MX575F512H. If you wish to apply the download package to a PIC32MX795F512L, all you have to do is inform the XC32 C compiler that the target is a PIC32MX795F512L instead of a PIC32MX575F512H.

The download package also contains the complete ExpressPCB hardware and schematic file set for the PIC32MX575F512H and PIC32MX795F512L printed circuit boards we have designed.

We still have some coding to do. The PIC32MX575F512H and PIC32MX795F512L drivers for the microSD card are still to be tapped out. We also need to feed some characters through the FTDI and ST3232 serial ports. The RTCC driver firmware is also waiting to be completed.

So, it looks like I'll be a busy little boy. We'll pick up where we left off in the next installment of the Design Cycle. **NV**

RESOURCES

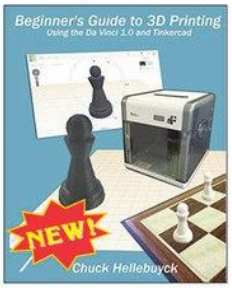
Microchip
PIC32MX575F512H
PIC32MX795F512L
MPLAB X
XC32 C Compiler
www.microchip.com

ExpressPCB
Printed circuit boards
www.expresspcb.com

Lemos International
ACKme Hopper
www.lemosint.com

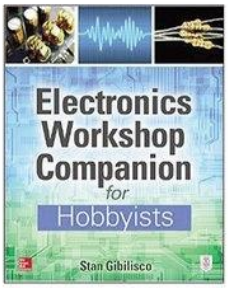
GREAT FOR DIYers!

Beginner's Guide to 3D Printing
by Chuck Hellebuyck



This book was written to answer the questions most beginners need answered. It covers many of the popular 3D printer choices and then uses the under $500 Da Vinci 1.0 from XYZprinting to show how easy it is to get started. Chuck takes you through using Tinkercad software for creating your own custom designs, then goes further and shows you how to take a simple design and send it off to a professional 3D printer. This book was designed for anyone just getting started. **$24.95**

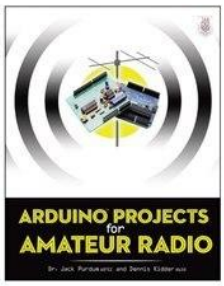
Electronics Workshop Companion for Hobbyists
by Stan Gibilisco



In this practical guide, electronics expert Stan Gibilisco shows you step-by-step how to set up a home workshop so you can invent, design, build, test, and repair electronic circuits and gadgets. *Electronics Workshop Companion for Hobbyists* provides tips for constructing your workbench and stocking it with the tools, components, and test equipment you'll need. Clear illustrations and interesting do-it-yourself experiments are included throughout this hands-on resource. **$25.00**

Arduino Projects for Amateur Radio
by Jack Purdum, Dennis Kidder

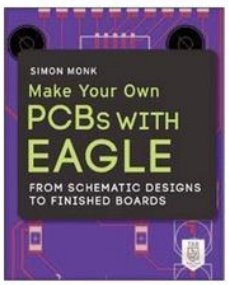
Boost Your Ham Radio's Capabilities Using Low Cost Arduino Microcontroller Boards



Do you want to increase the functionality and value of your ham radio without spending a lot of money? This book will show you how! *Arduino Projects for Amateur Radio* is filled with step-by-step microcontroller projects you can accomplish on your own — no programming experience necessary.

Reg Price $30.00 Sale Price $24.00

Make Your Own PCBs with EAGLE
by Eric Kleinert

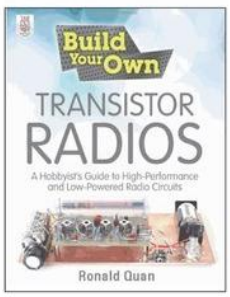


Featuring detailed illustrations and step-by-step instructions, *Make Your Own PCBs with EAGLE* leads you through the process of designing a schematic and transforming it into a PCB layout. You'll then move on to fabrication via the generation of standard Gerber files for submission to a PCB manufacturing service. This practical guide offers an accessible, logical way to learn EAGLE and start producing PCBs as quickly as possible.

$30.00

Build Your Own Transistor Radios
by Ronald Quan

A Hobbyist's Guide to High Performance and Low-Powered Radio Circuits

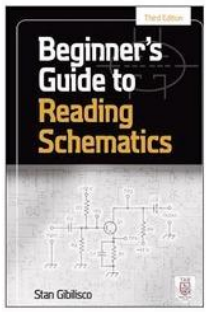


Create sophisticated transistor radios that are inexpensive yet highly efficient. Inside this book, it offers complete projects with detailed schematics and insights on how the radios were designed. Learn how to choose components, construct the different types of radios, and troubleshoot your work.

***Paperback, 496 pages**

Reg Price $49.95 Sale Price $39.95

Beginner's Guide to Reading Schematics, 3E
by Stan Gibilisco



Navigate the roadmaps of simple electronic circuits and complex systems with help from an experienced engineer. With all-new art and demo circuits you can build, this hands-on, illustrated guide explains how to understand and create high-precision electronics diagrams. Find out how to identify parts and connections, decipher element ratings, and apply diagram-based information in your own projects.

$25.00

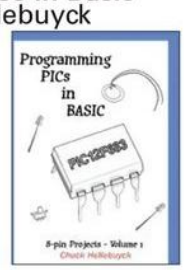
How to Diagnose and Fix Everything Electronic
by Michael Jay Geier

Master the Art of Electronics Repair

In this hands-on guide, a lifelong electronics repair guru shares his tested techniques and invaluable insights. *How to Diagnose and Fix Everything Electronic* shows you how to repair and extend the life of all kinds of solid-state devices, from modern digital gadgetry to cherished analog products of yesteryear.

$24.95

Programming PICs in Basic
by Chuck Hellebuyck

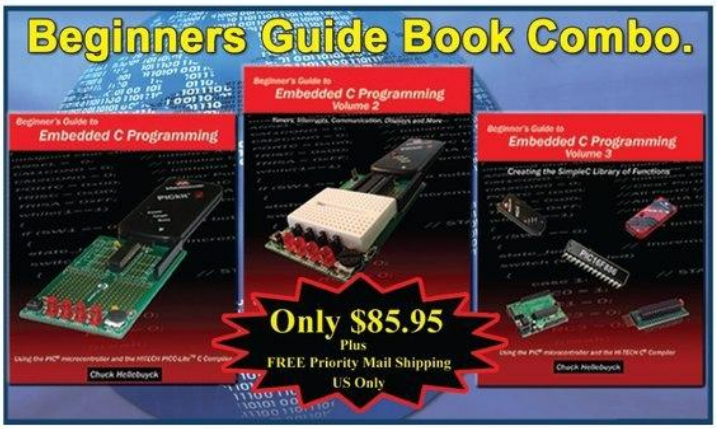


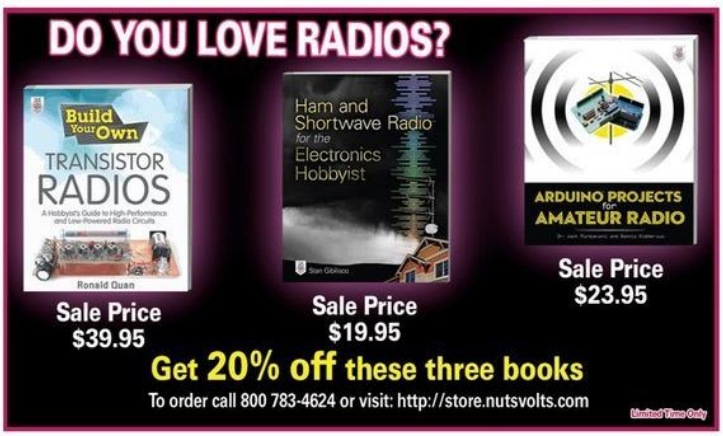
If you wanted to learn how to program microcontrollers, then you've found the right book! Microchip PIC microcontrollers are being designed into electronics throughout the world and none is more popular than the eight-pin version. Now the home hobbyist can create projects with these little microcontrollers using a low cost development tool called the CHIPAXE system and the Basic software language. Chuck Hellebuyck introduces how to use this development setup to build useful projects with an eight-pin PIC12F683 microcontroller. **$14.95**

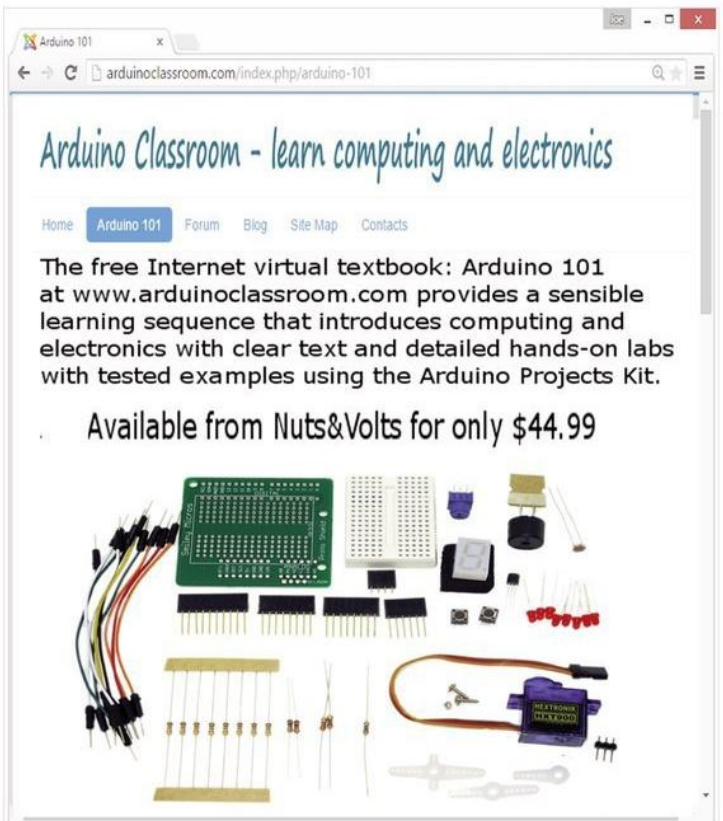
Programming Arduino Next Steps: Going Further with Sketches
by Simon Monk

In this practical guide, electronics guru Simon Monk takes you under the hood of Arduino and reveals professional programming secrets. Also shows you how to use interrupts, manage memory, program for the Internet, maximize serial communications, perform digital signal processing, and much more. All of the 75+ example sketches featured in the book are available for download. **$20.00**

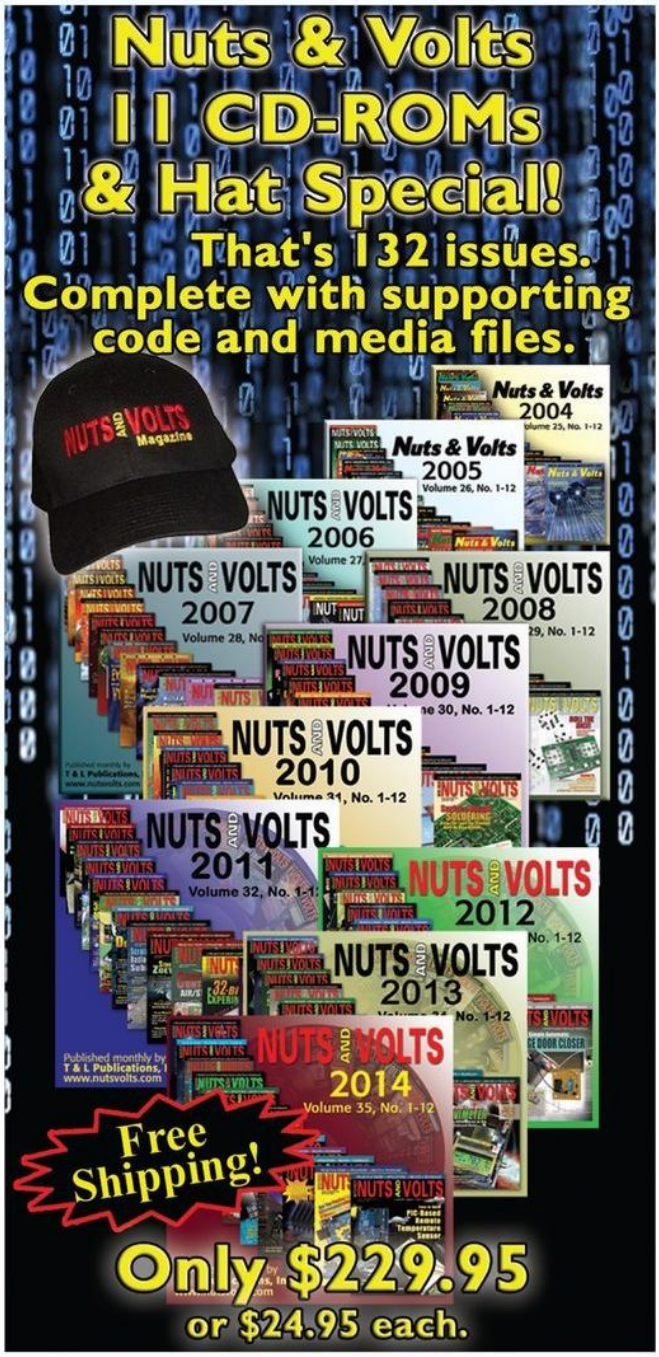
EDUCATIONAL

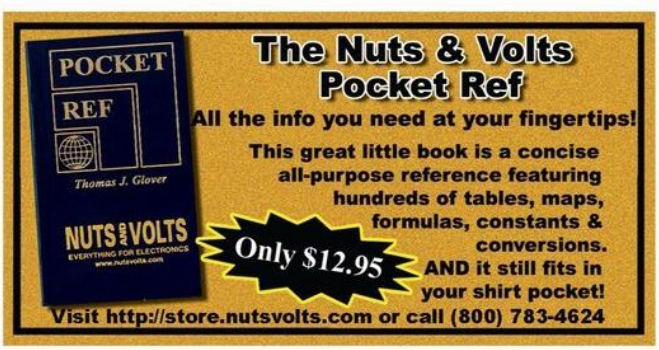






CD-ROM SPECIAL





PROJECTS

Fading Eyes Deluxe Board

The fading eyes circuit gives you two LED eyes that can be adjusted between a slow fade-in/fade-out to quick pulses. The speed is changed by simply adjusting the variable resistor with a small screw driver. Another adjustment allows you to set how long the LED stays on within the on/off fade cycle. Includes a battery snap and two red LEDs and two 24 inch eye cables.

$16.95

Talking Skull Kit

It's back!

The new and improved Talking Skull Kit. This latest version includes an improved audio and servo driver board.
Go to our webstore online to see all the improvements firsthand.
This kit provides everything necessary to build one talking skull. You provide the labor and tools, and you'll have a great Halloween prop in no time!

$97.95

Peek-a-Boo Ghost Kit

The Peek-a-Boo Ghost kit is a fun, low cost multi-use microcontroller kit. When triggered by the included motion sensor, this mini animatronic waves its arms, lights its LED eyes, and plays back the sounds you record. Perfect for kids, this kit can be used to create a fun Halloween prop for your desk, front door, or walkway. Watch the video to see this cool kit in action. Available in both a program-it-yourself or with a pre-programmed PICAXE chip option.

Un-programmed Chip Kit **$29.95**
Pre-programmed Chip Kit **$37.95**

Solar Charge Controller Kit 2.0

If you charge batteries using solar panels, then you can't afford not to have them protected from over-charging. This 12 volt/12 amp charge controller is great protection for the money. It is simple to build, ideal for the novice, and no special tools are needed other than a soldering iron and a 9/64" drill!

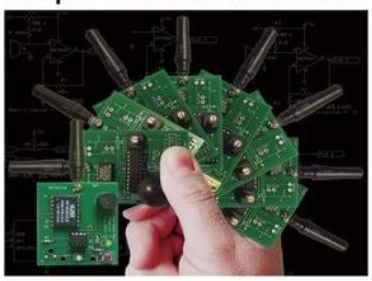
$27.95

Geiger Counter Kit

This kit is a great project for high school and university students The unit detects and displays levels of radiation, and can detect and display dosage levels as low as one micro-roentgen/hr. The LND 712 tube in our kit is capable of measuring alpha, beta, and gamma particles.
Partial kits also available.

$159.95

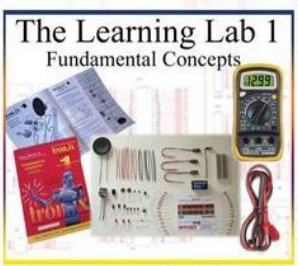
Super Detector Circuit Set



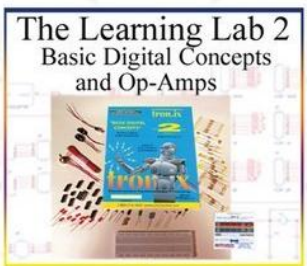
Pick a circuit!
With one PCB you have the option of detecting wirelessly: temperature, vibration, light, sound, motion, normally open switch, normally closed switch, any varying resistor input, voltage input, mA input, and tilt, just to name a few.

$32.95

FOR BEGINNER GEEKS!



$59.95



$49.95

$39.95

These labs from LF Components show simple and interesting experiments and lessons, all done on a solderless circuit board.

As you do each experiment, you learn how basic components work in a circuit, and continue to build your arsenal of knowledge with each successive experiment.

For more info and lab details, please visit our webstore.

Continued from page 20

BNC FEED-THROUGH TERMINATOR UPGRADE

Cal Test Electronics has announced an upgrade to its popular 50Ω BNC feed-through terminator, along with the introduction of two new models with impedances of 75Ω and 93Ω. The improved model CT2944 — now designated CT2944-50 — features an increased 2 GHz bandwidth and an impedance tolerance of ±0.25Ω. In addition, its accuracy is now 0.5%.

The CT2944's BNC female coaxial connector is gold plated for lower contact resistance and corrosion protection. Use the improved CT2944-50 to match a 50Ω coaxial cable to a 1 MΩ oscilloscope, a 1 MΩ probe to a 50Ω input instrument, or anywhere impedance matching is a must for measurement accuracy.

Along with the upgraded 50Ω version, the BNC feed-through terminator family now includes 75Ω and 93Ω models. The CT2944-75 and CT2944-93 are useful where cables of matching impedances are being used. This would include the popular RG59 and RG62 coaxial cables, respectively. The new models feature 1 GHz bandwidth and 1W average power rating along with the gold plated BNC female connector. Accuracy is also 0.5% for both.

The CT2944-50 ($30) is available now. Both the CT2944-75 ($52) and -93 ($52) will be available as of September 1, 2015.

For more information, contact:
Cal Test Electronics
www.caltestelectronics.com

CLASSIFIEDS

SURPLUS

WIRE/CABLE

ROBOTICS

HARDWARE WANTED

LIGHTING

SERVICES

READER - TO - READER TECHFORUM

All New TECHFORUM Online At www.nutsvolts.com/tech-forum

>>> QUESTIONS

Speaker Re-Coning

I have an older pair of Cerwin-Vega speakers where the foam edge has rotted and fallen apart on the woofers. I would love advice on the pros and cons of "re-coning" vs. buying new woofers. The model of the speaker is "R-24."

#9151 **Daniel Cook**
Palm Coast, FL

Is It Magic?

I have satellite radio (Sirius) aux audio output connected to DLO TransPod aux input. The TransPod is normally used for iPod to FM radio in a car. I am using it in my house.

The audio is transmitted on a frequency to my FM radio (50 ft away) with no power to the TransPod! How is this possible?

#9152 **Dennis Hole**
via email

Geocache Container With Flashing Light

I'm building a geocache container. To make it look authentic, I'd like the red light to flash every few seconds. Unfortunately, I don't have any "Nuts & Volts" electronics skills — what type of long-lasting battery and low-power LED light to use — and am hoping to get some expert advice.

#9153 **Steve Bass**
Altadena, CA

>>> ANSWERS

Tie Breaker Circuit

Does anyone have a circuit for a homemade "tie breaker" system? A teacher at my son's school is having a quiz contest where kids have to "buzz in." I need to build a circuit that can indicate who pressed a button first. I would prefer to use simple electrical components for this project as I am not really adept at programming microcontrollers.

The attached circuits (**Figures 1** and **2**) should do the trick. Each student station is equipped with a normally-open pushbutton switch connected to the instructor's console via a cable of suitable length, terminating at a connector.

The instructor's console consists of a bank of LEDs, one per student; a Master Reset pushbutton switch; and a suitable number of two-pin connectors into which the student pushbutton cables are plugged.

Each student readout consists of a "D" flipflop, an AND gate, an inverter, a diode, and an LED, plus associated wires, resistors, and capacitors as shown. The LED can be red or green, of any physical shape, having a maximum continuous current rating of 20 mA. The circuit is set for about 10 mA through the LED, which is very conservative and will provide quite adequate illumination.

The CMOS 4013B contains two "D" flipflops as shown, in a 14-pin package; power pins are 14(+) and 7(-). The CMOS 4081B contains four two-input AND gates as shown, in a 14-pin package; power pins are 14(+) and 7(-). The CMOS 4049B contains six inverters as shown, in a 16-pin package; power pins are 1(+) and 8(-). Be careful of this — applying "+" power to pin 1 is unconventional, but this is the way that the package is designed.

The circuit works in the following manner:

- Assume that the Master Reset button has been pushed on the teacher's console, asserting the +RESET bus. This forces all "D" flops to be reset, and all LEDs to be extinguished. The voltage level on the (diode-OR'd) -INHIBIT bus applies a logic-high signal to the D inputs of all flipflops, as well as to one of the inputs of each of the AND gates.
- The first student to press a button enables the AND gate for her/his receiver, which sets his/her "D" flipflop to the "on" state. Immediately its Q-bar output goes low, pulling down (asserting) the -INHIBIT bus, which presents a logic-low signal to the D inputs of all flipflops and disables all of the AND gates, thereby inhibiting further clocking pulses to all "D" flipflops.

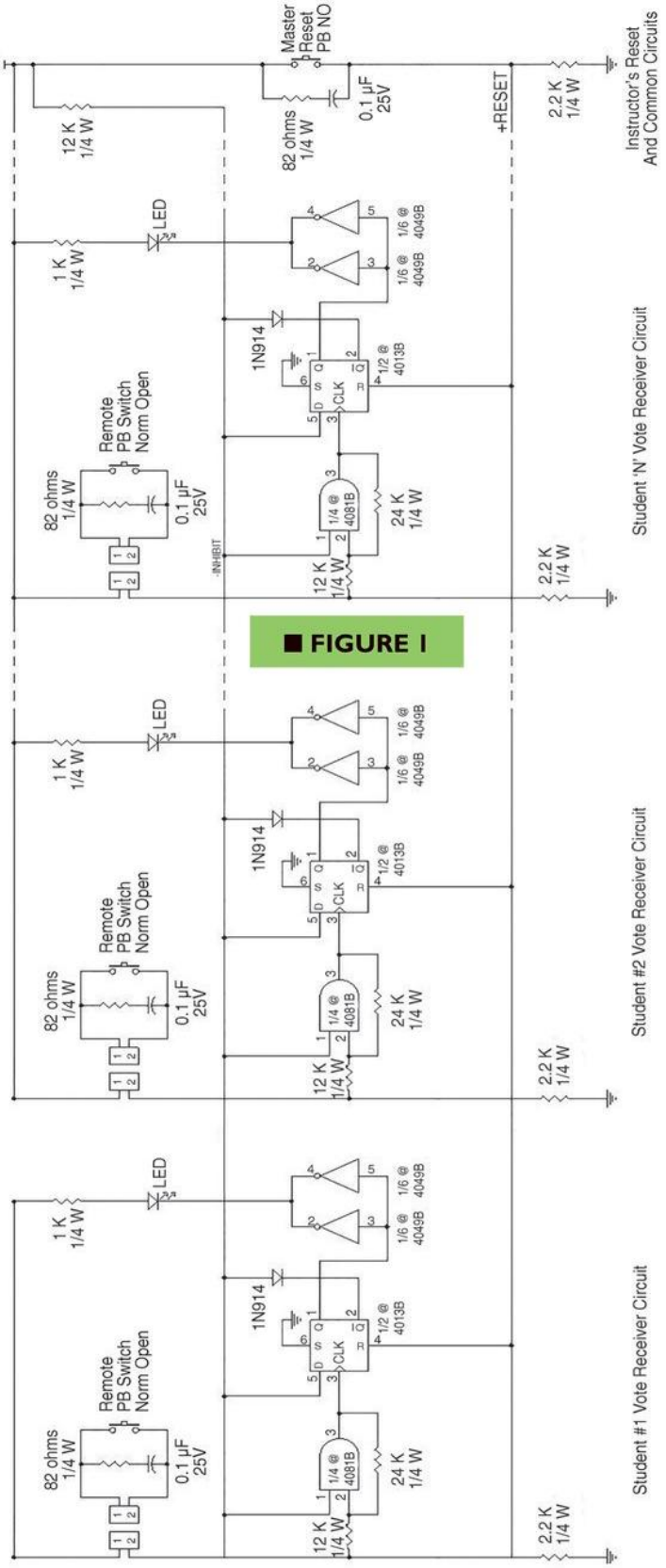
This ensures two things:

(a) that subsequent button depressions by other students cannot set the "D" flipflop in their receiver circuits to the "on" state; and

(b) that a second depression of the winning student's pushbutton will not set the "D" flipflop in her/his receiver to the "off" state, thereby negating her/his vote and inadvertently opening the process to a new round of voting.

The two resistors shown relative to the output and one input of each AND gate provides hysteresis and reduces noise sensitivity. As implemented, the input signal delivered to the 12K ohm resistor must rise above eight volts or fall below 4.5 volts before a high or low output level (respectively) will be

Send all questions and answers by email to **forum@nutsvolts.com**
or via the online form at **www.nutsvolts.com/tech-forum**



■ **FIGURE 1**

realized at the gate output (assuming that the remaining input to the AND gate is held high).

The contacts in each pushbutton are debounced with a paralleled R-C network.

The 12 VDC power can be provided by a small wallwart regulated power supply. Anything that will provide 100 mA or more will suffice.

Construction Suggestions:

1. Note that the integrated circuit (IC) package schematics show the two 4013B elements, the six 4049B elements, and the four 4081B elements, and that each IC is bypassed by a 0.01 µF disk ceramic capacitor, to be connected with the shortest possible leads across the power input terminals of the IC.

2. Pin assignments in the circuit schematic are illustrative; consult the IC package schematics for actual pin-outs for each circuit element in any particular IC.

3. The tie-breaker circuit shows two 4049B elements connected in parallel. Both elements must reside in the same IC package.

4. Diodes are 1N914 or 1N4148 signal diodes. All resistors are 1/4 watt carbon composition. With one exception, the capacitors are all disk ceramic, of any type (Z5U or better), 25 volts rating. The polarized capacitor is tantalum, and is to be connected directly across the 12V power input to the circuit board.

5. While the circuit schematic shows only three "student receiver" circuits, there does not seem to be any reasonable limit to the number that might be included, so long as +12V, circuit ground, and the +RESET and -INHIBIT busses are extended to all.

6. The input pins of all unused circuit elements must be connected to ground.

7. The student switches should be connected to the main console using twisted-pair cable to ensure minimum noise sensitivity. Mount the R-C debounce circuit directly across the pushbutton pins.

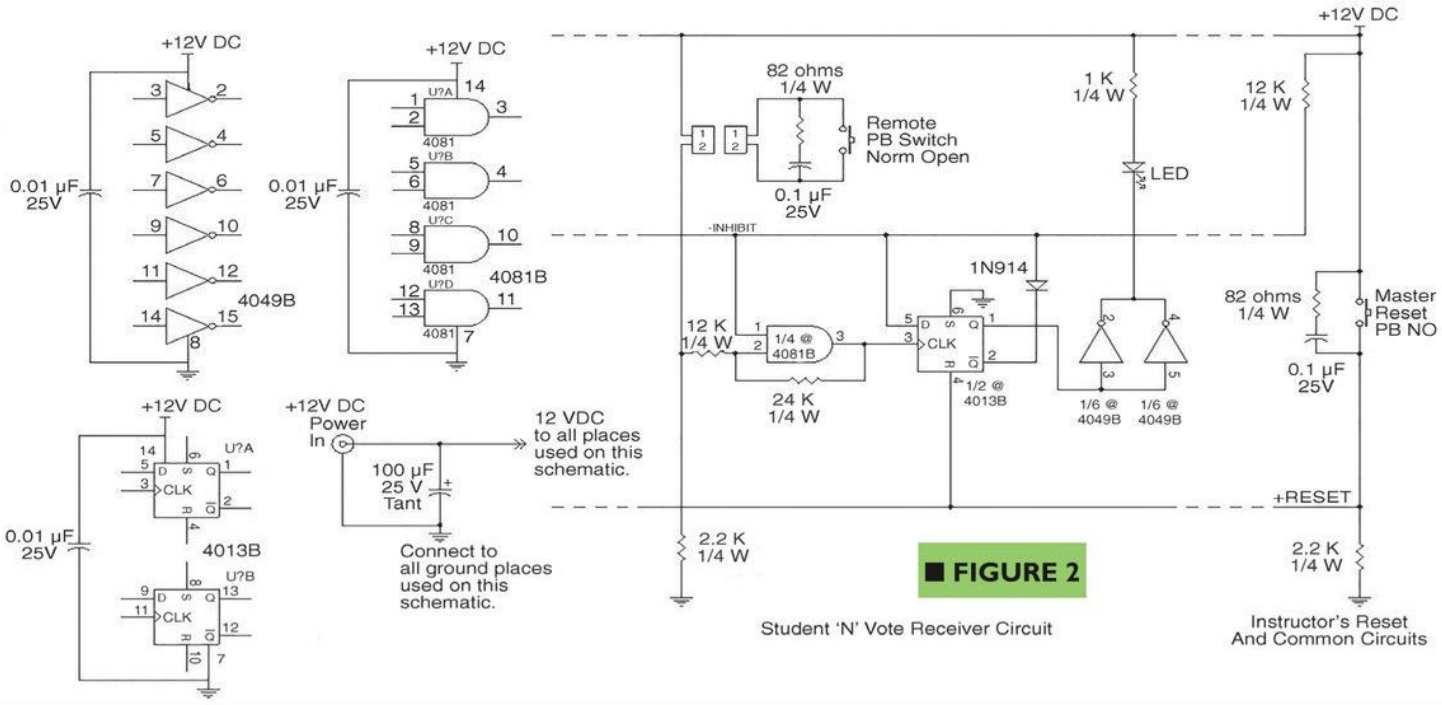
8. Dual-in-line package (DIP)-type ICs and IC sockets are recommended.

Datasheets for the three ICs may be found at:
www.nxp.com/documents/data_sheet/HEF4013B.pdf
www.nxp.com/documents/data_sheet/HEF4049B.pdf
www.nxp.com/documents/data_sheet/HEF4081B.pdf

A 1N914 diode datasheet is located at:
www.vishay.com/docs/85622/1n914.pdf

Parts are all readily accessible from Digi-Key, Mouser, Jones Associates, Newark, etc.

Peter A. Goodwin
Rockport, MA



■ FIGURE 2

■LOOK FOR ■SEARCH FOR ■FIND

Find your favorite advertisers here!

ADvertiser INDEX

MaxBotix Inc., is one of America's fastest growing companies

MaxBotix®
High Performance Ultrasonic Rangefinders

Like

CE RoHS

www.maxbotix.com

info@maxbotix.com

PYRAMID OF POWER

7.4V SPECS

HSB-9360TH
Torque: 236 oz-in
Speed: 0.06 sec/60°

HSB-9370TH
Torque: 347 oz-in
Speed: 0.10 sec/60°

HSB-9380TH
Torque: 472 oz-in
Speed: 0.14 sec/60°

HSB-9465SH
Torque: 187 oz-in
Speed: 0.07 sec/60°

HSB-9475SH
Torque: 269 oz-in
Speed: 0.10 sec/60°

HSB-9485SH
Torque: 361 oz-in
Speed: 0.15 sec/60°

ENERGY EFFICIENT BRUSHLESS SERVOS

The Building Blocks of Ultimate Power and Performance!

When we set out to engineer our line of digital, brushless servos, we had greatness in mind. The kind of greatness inspired and expected by kings. Designed with high-end brushless motors this series of servos delivers the power required for all your engineering needs. The energy efficiency, low current consumption and consistent torque provide the solid foundation for the performance you deserve!

HITEC

Hitec RCD USA Inc
12115 Paine St. Poway, CA 92064
(858) 748-6948 / www.hitecrcd.com